PIPELINE ACCELERATOR AND RELATED SYSTEM AND METHOD

Publication number:	EP1573515 (A2)	**Also published as:**	
Publication date:	2005-09-14		EP1573515 (A3)
Inventor(s):	RAPP JOHN W [US]; JACKSON LARRY [US]; JONES MARK [US]; CHERASARO TROY [US]		WO2004042560 (A2)
			WO2004042560 (A3)
Applicant(s):	LOCKHEED CORP [US]		WO2004042574 (A2)
Classification:			WO2004042574 (A3)
- international:	*G06F9/30; G06F9/38; G06F9/445; G06F9/46; G06F15/78;*		
	G06F9/30; G06F9/38; G06F9/445; G06F9/46; G06F15/76;	more >>	
	(IPC1-7): G06F9/38; G06F15/78		
- European:	G06F9/38S4		
Application number:	EP20030781553 20031031		
Priority number(s):	WO2003US34558 20031031; US20020422503P 20021031; US20030683929 20031009; US20030683932 20031009; US20030684053 20031009; US20030684057 20031009; US20030684102 20031009		

Abstract not available for EP 1573515 (A2)
Abstract of corresponding document: **WO 2004042560 (A2)**

A peer-vector machine includes a host processor and a hardwired pipeline accelerator. The host processor executes a program, and, in response to the program, generates host data, and the pipeline accelerator generates pipeline data from the host data. Alternatively, the pipeline accelerator generates the pipeline data, and the host processor generates the host data from the pipeline data. Because the peer-vector machine includes both a processor and a pipeline accelerator, it can often process data more efficiently than a machine that includes only processors or only accelerators. For example, one can design the peer-vector machine so that the host processor performs decision-making and non-mathematically intensive operations and the accelerator performs non-decision-making and mathematically intensive operations ; By shifting the mathematically intensive operations to the accelerator, the peer-vector machine often can, for a given clock frequency, process data at a speed that surpasses the speed at which a processor-only machine can process the data.

Internationale Anmeldung veröffentlicht durch die Weltorganisation für geistiges Eigentum unter der Nummer:

WO 2004/042562 (art. 158 des EPÜ).

International application published by the World Intellectual Property Organisation under number:

WO 2004/042562 (art. 158 of the EPC).

Demande internationale publiée par l'Organisation Mondiale de la Propriété sous le numéro:

WO 2004/042562 (art. 158 de la CBE).

(54) Title: PIPELINE ACCELERATOR FOR IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD

(57) Abstract: A pipeline accelerator includes a memory and a hardwired-pipeline circuit coupled to the memory. The hardwired-pipeline circuit is operable to receive data, load the data into the memory, retrieve the data from the memory, process the retrieved data, and provide the processed data to an external source. In addition or in the alternative, the hardwired-pipeline circuit is operable to receive data, process the received data, load the processed data into the memory, retrieve the processed data from the memory, and provide the retrieved processed data to an external source. Where the pipeline accelerator is coupled to a processor as part of a peer-vector machine, the memory facilitates the transfer of data – whether unidirectional or bidirectional – between the hardwired-pipeline circuit(s) and an application that the processor executes.

PIPELINE ACCELERATOR FOR IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD

CLAIM OF PRIORITY

[1]   This application claims priority to U.S. Provisional Application Serial
5   No. 60/422,503, filed on October 31, 2002, which is incorporated by reference.

CROSS REFERENCE TO RELATED APPLICATIONS

[2]   This application is related to U.S. Patent App. Serial Nos. 10/684,102
entitled IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND
METHOD, 10/684,053 entitled COMPUTING MACHINE HAVING IMPROVED
10   COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD,
10/684,057 entitled PROGRAMMABLE CIRCUIT AND RELATED COMPUTING
MACHINE AND METHOD, and 10/683,932 entitled PIPELINE ACCELERATOR
HAVING MULTIPLE PIPELINE UNITS AND RELATED COMPUTING MACHINE
AND METHOD, all filed on October 9, 2003, and having a common owner, and
15   which are incorporated by reference.

BACKGROUND

[3]   A common computing architecture for processing relatively large
amounts of data in a relatively short period of time includes multiple interconnected
processors that share the processing burden. By sharing the processing burden,
20   these multiple processors can often process the data more quickly than a single
processor can for a given clock frequency. For example, each of the processors can
process a respective portion of the data or execute a respective portion of a
processing algorithm.

[4]   FIG. 1 is a schematic block diagram of a conventional computing
25   machine *10* having a multi-processor architecture. The machine *10* includes a
master processor *12* and coprocessors *14₁ – 14ₙ*, which communicate with each
other and the master processor via a bus *16*, an input port *18* for receiving raw data
from a remote device (not shown in **FIG. 1**), and an output port *20* for providing
processed data to the remote source. The machine *10* also includes a memory *22*
30   for the master processor *12*, respective memories *24₁ – 24ₙ* for the coprocessors *14₁*

1

– 14_n, and a memory **26** that the master processor and coprocessors share via the bus **16**. The memory **22** serves as both a program and a working memory for the master processor **12**, and each memory $24_1 - 24_n$ serves as both a program and a working memory for a respective coprocessor $14_1 - 14_n$. The shared memory **26**

5   allows the master processor **12** and the coprocessors **14** to transfer data among themselves, and from/to the remote device via the ports **18** and **20**, respectively. The master processor **12** and the coprocessors **14** also receive a common clock signal that controls the speed at which the machine **10** processes the raw data.

[5]   In general, the computing machine **10** effectively divides the

10   processing of raw data among the master processor **12** and the coprocessors **14**. The remote source (not shown in **FIG. 1**) such as a sonar array loads the raw data via the port **18** into a section of the shared memory **26**, which acts as a first-in-first-out (FIFO) buffer (not shown) for the raw data. The master processor **12** retrieves the raw data from the memory **26** via the bus **16**, and then the master

15   processor and the coprocessors **14** process the raw data, transferring data among themselves as necessary via the bus **16**. The master processor **12** loads the processed data into another FIFO buffer (not shown) defined in the shared memory **26**, and the remote source retrieves the processed data from this FIFO via the port **20**.

20   [6]   In an example of operation, the computing machine **10** processes the raw data by sequentially performing n + 1 respective operations on the raw data, where these operations together compose a processing algorithm such as a Fast Fourier Transform (FFT). More specifically, the machine **10** forms a data-processing pipeline from the master processor **12** and the coprocessors **14**. For a given

25   frequency of the clock signal, such a pipeline often allows the machine **10** to process the raw data faster than a machine having only a single processor.

[7]   After retrieving the raw data from the raw-data FIFO (not shown) in the memory **26**, the master processor **12** performs a first operation, such as a trigonometric function, on the raw data. This operation yields a first result, which the

30   processor **12** stores in a first-result FIFO (not shown) defined within the memory **26**. Typically, the processor **12** executes a program stored in the memory **22**, and performs the above-described actions under the control of the program. The

processor **12** may also use the memory **22** as working memory to temporarily store data that the processor generates at intermediate intervals of the first operation.

[8]   Next, after retrieving the first result from the first-result FIFO (not shown) in the memory **26**, the coprocessor **14₁** performs a second operation, such

5   as a logarithmic function, on the first result. This second operation yields a second result, which the coprocessor **14₁** stores in a second-result FIFO (not shown) defined within the memory **26**. Typically, the coprocessor **14₁** executes a program stored in the memory **24₁**, and performs the above-described actions under the control of the program. The coprocessor **14₁** may also use the memory **24₁** as working memory to

10   temporarily store data that the coprocessor generates at intermediate intervals of the second operation.

[9]   Then, the coprocessors **24₂ – 24ₙ** sequentially perform third – nth operations on the second – (n-1)th results in a manner similar to that discussed above for the coprocessor **24₁**.

15   [10]   The nth operation, which is performed by the coprocessor **24ₙ**, yields the final result, *i.e.*, the processed data. The coprocessor **24ₙ** loads the processed data into a processed-data FIFO (not shown) defined within the memory **26**, and the remote device (not shown in **FIG. 1**) retrieves the processed data from this FIFO.

[11]   Because the master processor **12** and coprocessors **14** are

20   simultaneously performing different operations of the processing algorithm, the computing machine **10** is often able to process the raw data faster than a computing machine having a single processor that sequentially performs the different operations. Specifically, the single processor cannot retrieve a new set of the raw data until it performs all n + 1 operations on the previous set of raw data. But using

25   the pipeline technique discussed above, the master processor **12** can retrieve a new set of raw data after performing only the first operation. Consequently, for a given clock frequency, this pipeline technique can increase the speed at which the machine **10** processes the raw data by a factor of approximately n + 1 as compared to a single-processor machine (not shown in **FIG. 1**).

30   [12]   Alternatively, the computing machine **10** may process the raw data in parallel by simultaneously performing n + 1 instances of a processing algorithm,

such as an FFT, on the raw data. That is, if the algorithm includes n + 1 sequential operations as described above in the previous example, then each of the master processor 12 and the coprocessors 14 sequentially perform all n + 1 operations on respective sets of the raw data. Consequently, for a given clock frequency, this

5   parallel-processing technique, like the above-described pipeline technique, can increase the speed at which the machine 10 processes the raw data by a factor of approximately n + 1 as compared to a single-processor machine (not shown in **FIG. 1**).

[13]   Unfortunately, although the computing machine 10 can process data

10   more quickly than a single-processor computer machine (not shown in **FIG. 1**), the data-processing speed of the machine 10 is often significantly less than the frequency of the processor clock. Specifically, the data-processing speed of the computing machine 10 is limited by the time that the master processor 12 and coprocessors 14 require to process data. For brevity, an example of this speed

15   limitation is discussed in conjunction with the master processor 12, although it is understood that this discussion also applies to the coprocessors 14. As discussed above, the master processor 12 executes a program that controls the processor to manipulate data in a desired manner. This program includes a sequence of instructions that the processor 12 executes. Unfortunately, the processor 12

20   typically requires multiple clock cycles to execute a single instruction, and often must execute multiple instructions to process a single value of data. For example, suppose that the processor 12 is to multiply a first data value A (not shown) by a second data value B (not shown). During a first clock cycle, the processor 12 retrieves a multiply instruction from the memory 22. During second and third clock

25   cycles, the processor 12 respectively retrieves A and B from the memory 26. During a fourth clock cycle, the processor 12 multiplies A and B, and, during a fifth clock cycle, stores the resulting product in the memory 22 or 26 or provides the resulting product to the remote device (not shown). This is a best-case scenario, because in many cases the processor 12 requires additional clock cycles for overhead tasks

30   such as initializing and closing counters. Therefore, at best the processor 12 requires five clock cycles, or an average of 2.5 clock cycles per data value, to process A and B..

[14]   Consequently, the speed at which the computing machine *10* processes data is often significantly lower than the frequency of the clock that drives the master processor *12* and the coprocessors *14*. For example, if the processor *12* is clocked at 1.0 Gigahertz (GHz) but requires an average of 2.5 clock cycles per data value, then the effective data-processing speed equals (1.0 GHz)/2.5 = 0.4 GHz. This effective data-processing speed is often characterized in units of operations per second. Therefore, in this example, for a clock speed of 1.0 GHz, the processor *12* would be rated with a data-processing speed of 0.4 Gigaoperations/second (Gops).

[15]   FIG. 2 is a block diagram of a hardwired data pipeline *30* that can typically process data faster than a processor can for a given clock frequency, and often at substantially the same rate at which the pipeline is clocked. The pipeline *30* includes operator circuits $32_1 - 32_n$, which each perform a respective operation on respective data without executing program instructions. That is, the desired operation is "burned in" to a circuit *32* such that it implements the operation automatically, without the need of program instructions. By eliminating the overhead associated with executing program instructions, the pipeline *30* can typically perform more operations per second than a processor can for a given clock frequency.

[16]   For example, the pipeline *30* can often solve the following equation faster than a processor can for a given clock frequency:

$$Y(x_k) = (5x_k + 3)2^{x_k}$$

where x_k represents a sequence of raw data values. In this example, the operator circuit 32_1 is a multiplier that calculates $5x_k$, the circuit 32_2 is an adder that calculates $5x_k + 3$, and the circuit 32_n (n = 3) is a multiplier that calculates $(5x_k + 3)2^{x_k}$.

[17]   During a first clock cycle k=1, the circuit 32_1 receives data value x_1 and multiplies it by 5 to generate $5x_1$.

[18]   During a second clock cycle k = 2, the circuit 32_2 receives $5x_1$ from the circuit 32_1 and adds 3 to generate $5x_1 + 3$. Also, during the second clock cycle, the circuit 32_1 generates $5x_2$.

[19]   During a third clock cycle k = 3, the circuit 32_3 receives $5x_1 + 3$ from the circuit 32_2 and multiplies by 2^{x_1} (effectively left shifts $5x_1 + 3$ by x_1) to generate

the first result $(5x_1 + 3)2^{x1}$. Also during the third clock cycle, the circuit 32_1 generates $5x_3$ and the circuit 32_2 generates $5x_2 + 3$.

[20]   The pipeline 30 continues processing subsequent raw data values x_k in this manner until all the raw data values are processed.

[21]   Consequently, a delay of two clock cycles after receiving a raw data value x_1 — this delay is often called the latency of the pipeline 30 — the pipeline generates the result $(5x_1 + 3)2^{x1}$, and thereafter generates one result — e.g., $(5x_2 + 3)2^{x2}$, $(5x_3 + 3)2^{x3}$, . . ., $5x_n + 3)2^{xn}$ — each clock cycle.

[22]   Disregarding the latency, the pipeline 30 thus has a data-processing speed equal to the clock speed. In comparison, assuming that the master processor 12 and coprocessors 14 (FIG. 1) have data-processing speeds that are 0.4 times the clock speed as in the above example, the pipeline 30 can process data 2.5 times faster than the computing machine 10 (FIG. 1) for a given clock speed.

[23]   Still referring to FIG. 2, a designer may choose to implement the pipeline 30 in a programmable logic IC (PLIC), such as a field-programmable gate array (FPGA), because a PLIC allows more design and modification flexibility than does an application specific IC (ASIC). To configure the hardwired connections within a PLIC, the designer merely sets interconnection-configuration registers disposed within the PLIC to predetermined binary states. The combination of all these binary states is often called "firmware." Typically, the designer loads this firmware into a nonvolatile memory (not shown in FIG. 2) that is coupled to the PLIC. When one "turns on" the PLIC, it downloads the firmware from the memory into the interconnection-configuration registers. Therefore, to modify the functioning of the PLIC, the designer merely modifies the firmware and allows the PLIC to download the modified firmware into the interconnection-configuration registers. This ability to modify the PLIC by merely modifying the firmware is particularly useful during the prototyping stage and for upgrading the pipeline 30 "in the field".

[24]   Unfortunately, the hardwired pipeline 30 may not be the best choice to execute algorithms that entail significant decision making, particularly nested decision making. A processor can typically execute a nested-decision-making instruction (e.g., a nested conditional instruction such as "if A, then do B, else if C,

do D, . . ., else do n") approximately as fast as it can execute an operational
instruction (e.g., "A + B") of comparable length. But although the pipeline **30** may be
able to make a relatively simple decision (e.g., "A > B?") efficiently, it typically cannot
execute a nested decision (e.g., "if A, then do B, else if C, do D, . . ., else do n") as

5   efficiently as a processor can. One reason for this inefficiency is that the pipeline **30**
may have little on-board memory, and thus may need to access external
working/instruction memory (not shown). And although one may be able to design
the pipeline **30** to execute such a nested decision, the size and complexity of the
required circuitry often makes such a design impractical, particularly where an

10   algorithm includes multiple different nested decisions.

[25]   Consequently, processors are typically used in applications that
require significant decision making, and hardwired pipelines are typically limited to
"number crunching" applications that entail little or no decision making.

[26]   Furthermore, as discussed below, it is typically much easier for one to

15   design/modify a processor-based computing machine, such as the computing
machine **10** of **FIG. 1**, than it is to design/modify a hardwired pipeline such as the
pipeline **30** of **FIG. 2**, particularly where the pipeline **30** includes multiple PLICs.

[27]   Computing components, such as processors and their peripherals
(e.g., memory), typically include industry-standard communication interfaces that

20   facilitate the interconnection of the components to form a processor-based
computing machine.

[28]   Typically, a standard communication interface includes two layers: a
physical layer and a services layer.

[29]   The physical layer includes the circuitry and the corresponding circuit

25   interconnections that form the interface and the operating parameters of this
circuitry. For example, the physical layer includes the pins that connect the
component to a bus, the buffers that latch data received from the pins, and the
drivers that drive signals onto the pins. The operating parameters include the
acceptable voltage range of the data signals that the pins receive, the signal timing

30   for writing and reading data, and the supported modes of operation (e.g., burst

mode, page mode). Conventional physical layers include transistor-transistor logic (TTL) and RAMBUS.

[30]   The services layer includes the protocol by which a computing component transfers data. The protocol defines the format of the data and the

5   manner in which the component sends and receives the formatted data. Conventional communication protocols include file-transfer protocol (FTP) and transmission control protocol/internet protocol (TCP/IP).

[31]   Consequently, because manufacturers and others typically design computing components having industry-standard communication interfaces, one can

10   typically design the interface of such a component and interconnect it to other computing components with relatively little effort. This allows one to devote most of his time to designing the other portions of the computing machine, and to easily modify the machine by adding or removing components.

[32]   Designing a computing component that supports an industry-standard

15   communication interface allows one to save design time by using an existing physical-layer design from a design library. This also insures that he/she can easily interface the component to off-the-shelf computing components.

[33]   And designing a computing machine using computing components that support a common industry-standard communication interface allows the designer to

20   interconnect the components with little time and effort. Because the components support a common interface, the designer can interconnect them via a system bus with little design effort. And because the supported interface is an industry standard, one can easily modify the machine. For example, one can add different components and peripherals to the machine as the system design evolves, or can easily

25   add/design next-generation components as the technology evolves. Furthermore, because the components support a common industry-standard service layer, one can incorporate into the computing machine's software an existing software module that implements the corresponding protocol. Therefore, one can interface the components with little effort because the interface design is essentially already in

30   place, and thus can focus on designing the portions (e.g., software) of the machine that cause the machine to perform the desired function(s).

[34]   But unfortunately, there are no known industry-standard services layers for components, such as PLICs, used to form hardwired pipelines such as the pipeline *30* of **FIG. 2**.

[35]   Consequently, to design a pipeline having multiple PLICs, one typically

5   spends a significant amount of time and exerts a significant effort designing and debugging the services layer of the communication interface between the PLICs "from scratch." Typically, such an ad hoc services layer depends on the parameters of the data being transferred between the PLICs. Likewise, to design a pipeline that interfaces to a processor, one would have to spend a significant amount of time and

10   exert a significant effort in designing and debugging the services layer of the communication interface between the pipeline and the processor from scratch.

[36]   Similarly, to modify such a pipeline by adding a PLIC to it, one typically spends a significant amount of time and exerts a significant effort designing and debugging the services layer of the communication interface between the added

15   PLIC and the existing PLICs. Likewise, to modify a pipeline by adding a processor, or to modify a computing machine by adding a pipeline, one would have to spend a significant amount of time and exert a significant effort in designing and debugging the services layer of the communication interface between the pipeline and processor.

20   [37]   Consequently, referring to **FIGS. 1** and **2**, because of the difficulties in interfacing multiple PLICs and in interfacing a processor to a pipeline, one is often forced to make significant tradeoffs when designing a computing machine. For example, with a processor-based computing machine, one is forced to trade number-crunching speed and design/modification flexibility for complex decision-

25   making ability. Conversely, with a hardwired pipeline-based computing machine, one is forced to trade complex-decision-making ability and design/modification flexibility for number-crunching speed. Furthermore, because of the difficulties in interfacing multiple PLICs, it is often impractical for one to design a pipeline-based machine having more than a few PLICs. As a result, a practical pipeline-based

30   machine often has limited functionality. And because of the difficulties in interfacing a processor to a PLIC, it would be impractical to interface a processor to more than

one PLIC. As a result, the benefits obtained by combining a processor and a pipeline would be minimal.

[38]   Therefore, a need has arisen for a new computing architecture that allows one to combine the decision-making ability of a processor-based machine with the number-crunching speed of a hardwired-pipeline-based machine.

<div align="center">

SUMMARY

</div>

[39]   According to an embodiment of the invention, a pipeline accelerator includes a memory and a hardwired-pipeline circuit coupled to the memory. The hardwired-pipeline circuit is operable to receive data, load the data into the memory, retrieve the data from the memory, process the retrieved data, and provide the processed data to an external source.

[40]   According to another embodiment of the invention, the hardwired-pipeline circuit is operable to receive data, process the received data, load the processed data into the memory, retrieve the processed data from the memory, and provide the retrieved processed data to an external source.

[41]   Where the pipeline accelerator is coupled to a processor as part of a peer-vector machine, the memory facilitates the transfer of data — whether unidirectional or bidirectional — between the hardwired-pipeline circuit and an application that the processor executes.

<div align="center">

BRIEF DESCRIPTION OF THE DRAWINGS

</div>

[42]   FIG. 1 is a block diagram of a computing machine having a conventional multi-processor architecture.

[43]   FIG. 2 is a block diagram of a conventional hardwired pipeline.

[44]   FIG. 3 is a block diagram of a computing machine having a peer-vector architecture according to an embodiment of the invention.

[45]   FIG. 4 is a block diagram of the pipeline accelerator of FIG. 3 according to an embodiment of the invention.

[46]   FIG. 5 is a block diagram of the hardwired-pipeline circuit and the data memory of FIG. 4 according to an embodiment of the invention.

<div align="center">

10

</div>

[47]   FIG. 6 is a block diagram of the memory-write interfaces of the communication shell of FIG. 5 according to an embodiment of the invention.

[48]   FIG. 7 is a block diagram of the memory-read interfaces of the communication shell of FIG. 5 according to an embodiment of the invention.

[49]   FIG. 8 is a block diagram of the pipeline accelerator of FIG. 3 according to another embodiment of the invention.

[50]   FIG. 9 is a block diagram of the hardwired-pipeline circuit and the data memory of FIG. 8 according to an embodiment of the invention.

DETAILED DESCRIPTION

[51]   FIG. 3 is a schematic block diagram of a computing machine *40*, which has a peer-vector architecture according to an embodiment of the invention. In addition to a host processor *42*, the peer-vector machine *40* includes a pipeline accelerator *44*, which performs at least a portion of the data processing, and which thus effectively replaces the bank of coprocessors *14* in the computing machine *10* of FIG. 1. Therefore, the host-processor *42* and the accelerator *44* (or units thereof as discussed below) are "peers" that can transfer data vectors back and forth. Because the accelerator *44* does not execute program instructions, it typically performs mathematically intensive operations on data significantly faster than a bank of coprocessors can for a given clock frequency. Consequently, by combining the decision-making ability of the processor *42* and the number-crunching ability of the accelerator *44*, the machine *40* has the same abilities as, but can often process data faster than, a conventional computing machine such as the machine *10*. Furthermore, as discussed below, providing the accelerator *44* with a communication interface that is compatible with the communication interface of the host processor *42* facilitates the design and modification of the machine *40*, particularly where the processor's communication interface is an industry standard. And where the accelerator *44* includes multiple pipeline units (*e.g.*, PLIC-based circuits), providing each of these units with the same communication interface facilitates the design and modification of the accelerator, particularly where the communication interfaces are compatible with an industry-standard interface.

Moreover, the machine **40** may also provide other advantages as described below
and in the previously cited patent applications.

[52]   Still referring to **FIG. 3**, in addition to the host processor **42** and the
pipeline accelerator **44**, the peer-vector computing machine **40** includes a processor
5   memory **46**, an interface memory **48**, a bus **50**, a firmware memory **52**, an optional
raw-data input port **54**, a processed-data output port **58**, and an optional router **61**.

[53]   The host processor **42** includes a processing unit **62** and a message
handler **64**, and the processor memory **46** includes a processing-unit memory **66**
and a handler memory **68**, which respectively serve as both program and working
10   memories for the processor unit and the message handler. The processor memory
46 also includes an accelerator-configuration registry **70** and a
message-configuration registry **72**, which store respective configuration data that
allow the host processor **42** to configure the functioning of the accelerator **44** and the
format of the messages that the message handler **64** sends and receives.

15   [54]   The pipeline accelerator **44** is disposed on at least one PLIC (not
shown) and includes hardwired pipelines $74_1 - 74_n$, which process respective data
without executing program instructions. The firmware memory **52** stores the
configuration firmware for the accelerator **44**. If the accelerator **44** is disposed on
multiple PLICs, these PLICs and their respective firmware memories may be
20   disposed in multiple pipeline units (**FIG. 4**). The accelerator **44** and pipeline units
are discussed further below and in previously cited U.S. Patent App. Serial No.
10/683,932 entitled PIPELINE ACCELERATOR HAVING MULTIPLE PIPELINE
UNITS AND RELATED COMPUTING MACHINE AND METHOD. Alternatively, the
accelerator **44** may be disposed on at least one ASIC, and thus may have internal
25   interconnections that are unconfigurable. In this alternative, the machine **40** may
omit the firmware memory **52**. Furthermore, although the accelerator **44** is shown
including multiple pipelines **74**, it may include only a single pipeline. In addition,
although not shown, the accelerator **44** may include one or more processors such as
a digital-signal processor (DSP). Moreover, although not shown, the accelerator **44**
30   may include a data input port and/or a data output port.

[55]   The general operation of the peer-vector machine **40** is discussed in
previously cited U.S. Patent App. Serial No. 10/684,102 entitled IMPROVED

COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD, and the structure and operation of the pipeline accelerator **44** is discussed below in conjunction with **FIGS. 4 – 9**.

[56]   **FIG. 4** is a schematic block diagram of the pipeline accelerator **44** of

5   **FIG. 3** according to an embodiment of the invention.

[57]   The accelerator **44** includes one or more pipeline units **78**, each of which includes a pipeline circuit **80**, such as a PLIC or an ASIC. As discussed further below and in previously cited U.S. Patent App. Serial No. 10/683,932 entitled PIPELINE ACCELERATOR HAVING MULTIPLE PIPELINE UNITS AND RELATED

10 COMPUTING MACHINE AND METHOD, each pipeline unit **78** is a "peer" of the host processor **42** and of the other pipeline units of the accelerator **44**. That is, each pipeline unit **78** can communicate directly with the host processor **42** or with any other pipeline unit. Thus, this peer-vector architecture prevents data "bottlenecks" that otherwise might occur if all of the pipeline units **78** communicated through a

15 central location such as a master pipeline unit (not shown) or the host processor **42**. Furthermore, it allows one to add or remove peers from the peer-vector machine **40** (**FIG. 3**) without significant modifications to the machine.

[58]   The pipeline circuit **80** includes a communication interface **82**, which transfers data between a peer, such as the host processor **42** (**FIG. 3**), and the

20 following other components of the pipeline circuit: the hardwired pipelines **74₁-74ₙ** (**FIG. 3**) via a communication shell **84**, a controller **86**, an exception manager **88**, and a configuration manager **90**. The pipeline circuit **80** may also include an industry-standard bus interface **91**. Alternatively, the functionality of the interface **91** may be included within the communication interface **82**.

25 [59]   By designing the components of the pipeline circuit **80** as separate modules, one can often simplify the design of the pipeline circuit. That is, one can design and test each of these components separately, and then integrate them much like one does when designing software or a processor-based computing system (such as the system **10** of **FIG. 1**). In addition, one can save in a library (not

30 shown) hardware description language (HDL) that defines these components — particularly components, such as the communication interface **82**, that will probably be used frequently in other pipeline designs — thus reducing the design and test

time of future pipeline designs that use the same components. That is, by using the HDL from the library, the designer need not redesign previously implemented components "from scratch", and thus can focus his efforts on the design of components that were not previously implemented, or on the modification of
5   previously implemented components. Moreover, one can save in the library HDL that defines multiple versions of the pipeline circuit *80* or of the entire pipeline accelerator *44*, so that one can pick and choose among existing designs.

[60]   The communication interface *82* sends and receives data in a format recognized by the message handler *64* (FIG. 3), and thus typically facilitates the
10   design and modification of the peer-vector machine *40* (FIG. 3). For example, if the data format is an industry standard such as the Rapid I/O format, then one need not design a custom interface between the host processor *42* and the accelerator *44*. Furthermore, by allowing the pipeline circuit *80* to communicate with other peers, such as the host processor *42* (FIG. 3), via the pipeline bus *50* instead of via a
15   non-bus interface, one can change the number of pipeline units *78* by merely connecting or disconnecting them (or the circuit cards that hold them) to the pipeline bus instead of redesigning a non-bus interface from scratch each time a pipeline unit is added or removed.

[61]   The hardwired pipelines 74_1-74_n perform respective operations on data
20   as discussed above in conjunction with FIG. 3 and in previously cited U.S. Patent App. Serial No. 10/684,102 entitled IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD, and the communication shell *84* interfaces the pipelines to the other components of the pipeline circuit *80* and to circuits (such as a data memory *92* discussed below) external to the pipeline circuit.

25   [62]   The controller *86* synchronizes the hardwired pipelines 74_1-74_n and monitors and controls the sequence in which they perform the respective data operations in response to communications, *i.e.*, "events," from other peers. For example, a peer such as the host processor *42* may send an event to the pipeline unit *78* via the pipeline bus *50* to indicate that the peer has finished sending a block
30   of data to the pipeline unit and to cause the hardwired pipelines 74_1-74_n to begin processing this data. An event that includes data is typically called a message, and an event that does not include data is typically called a "door bell." Furthermore, as

discussed below in conjunction with **FIG. 5**, the pipeline unit **78** may also synchronize the pipelines **74₁-74ₙ** in response to a synchronization signal.

[63]   The exception manager **88** monitors the status of the hardwired pipelines **74₁-74ₙ**, the communication interface **82**, the communication shell **84**, the

5   controller **86**, and the bus interface **91**, and reports exceptions to the host processor **42** (**FIG. 3**). For example, if a buffer in the communication interface **82** overflows, then the exception manager **88** reports this to the host processor **42**. The exception manager may also correct, or attempt to correct, the problem giving rise to the exception. For example, for an overflowing buffer, the exception manager **88** may

10   increase the size of the buffer, either directly or via the configuration manager **90** as discussed below.

[64]   The configuration manager **90** sets the soft configuration of the hardwired pipelines **74₁-74ₙ**, the communication interface **82**, the communication shell **84**, the controller **86**, the exception manager **88**, and the interface **91** in

15   response to soft-configuration data from the host processor **42** (**FIG. 3**) — as discussed in previously cited U.S. Patent App. Serial No. 10/684,102 entitled IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD, the hard configuration denotes the actual topology, on the transistor and circuit-block level, of the pipeline circuit **80**, and the soft configuration denotes the

20   physical parameters (e.g., data width, table size) of the hard-configured components. That is, soft configuration data is similar to the data that can be loaded into a register of a processor (not shown in **FIG. 4**) to set the operating mode (e.g., burst-memory mode) of the processor. For example, the host processor **42** may send soft-configuration data that causes the configuration manager **90** to set the

25   number and respective priority levels of queues in the communication interface **82**. The exception manager **88** may also send soft-configuration data that causes the configuration manager **90** to, e.g., increase the size of an overflowing buffer in the communication interface **82**.

[65]   Still referring to **FIG. 4**, in addition to the pipeline circuit **80**, the pipeline

30   unit **78** of the accelerator **44** includes the data memory **92**, an optional communication bus **94**, and, if the pipeline circuit is a PLIC, the firmware memory **52** (**FIG. 3**).

[66]   The data memory 92 buffers data as it flows between another peer, such as the host processor 42 (FIG. 3), and the hardwired pipelines 74_1-74_n, and is also a working memory for the hardwired pipelines. The communication interface 82 interfaces the data memory 92 to the pipeline bus 50 (via the communication bus 94

5   and industry-standard interface 91 if present), and the communication shell 84 interfaces the data memory to the hardwired pipelines 74_1-74_n.

[67]   The industry-standard interface 91 is a conventional bus-interface circuit that reduces the size and complexity of the communication interface 82 by effectively offloading some of the interface circuitry from the communication

10   interface. Therefore, if one wishes to change the parameters of the pipeline bus 50 or router 61 (FIG. 3), then he need only modify the interface 91 and not the communication interface 82. Alternatively, one may dispose the interface 91 in an IC (not shown) that is external to the pipeline circuit 80. Offloading the interface 91 from the pipeline circuit 80 frees up resources on the pipeline circuit for use in, e.g.,

15   the hardwired pipelines 74_1-74_n and the controller 86. Or, as discussed above, the bus interface 91 may be part of the communication interface 82.

[68]   As discussed above in conjunction with FIG. 3, where the pipeline circuit 80 is a PLIC, the firmware memory 52 stores the firmware that sets the hard configuration of the pipeline circuit. The memory 52 loads the firmware into the

20   pipeline circuit 80 during the configuration of the accelerator 44, and may receive modified firmware from the host processor 42 (FIG. 3) via the communication interface 82 during or after the configuration of the accelerator. The loading and receiving of firmware is further discussed in previously cited U.S. Patent App. Serial No. 10/684,057 entitled PROGRAMMABLE CIRCUIT AND RELATED COMPUTING

25   MACHINE AND METHOD.

[69]   Still referring to FIG. 4, the pipeline circuit 80, data memory 92, and firmware memory 52 may be disposed on a circuit board or card 98, which may be plugged into a pipeline-bus connector (not shown) much like a daughter card can be plugged into a slot of a mother board in a personal computer (not shown). Although

30   not shown, conventional ICs and components such as a power regulator and a power sequencer may also be disposed on the card 98 as is known.

[70]   Further details of the structure and operation of the pipeline unit **78** are discussed below in conjunction with **FIG. 5**.

[71]   **FIG. 5** is a block diagram of the pipeline unit **78** of **FIG. 4** according to an embodiment of the invention. For clarity, the firmware memory **52** is omitted from

5   **FIG. 5**. The pipeline circuit **80** receives a master CLOCK signal, which drives the below-described components of the pipeline circuit either directly or indirectly. The pipeline circuit **80** may generate one or more slave clock signals (not shown) from the master CLOCK signal in a conventional manner. The pipeline circuit **80** may also a receive a synchronization signal SYNC as discussed below.

10   [72]   The data memory **92** includes an input dual-port-static-random-access memory (DPSRAM) **100**, an output DPSRAM **102**, and an optional working DPSRAM **104**.

[73]   The input DPSRAM **100** includes an input port **106** for receiving data from a peer, such as the host processor **42** (**FIG. 3**), via the communication interface

15   **82**, and includes an output port **108** for providing this data to the hardwired pipelines **74₁-74ₙ** via the communication shell **84**. Having two ports, one for data input and one for data output, increases the speed and efficiency of data transfer to/from the DPSRAM **100** because the communication interface **82** can write data to the DPSRAM while the pipelines **74₁-74ₙ** read data from the DPSRAM. Furthermore, as

20   discussed above, using the DPSRAM **100** to buffer data from a peer such as the host processor **42** allows the peer and the pipelines **74₁-74ₙ** to operate asynchronously relative to one and other. That is, the peer can send data to the pipelines **74₁-74ₙ** without "waiting" for the pipelines to complete a current operation. Likewise, the pipelines **74₁-74ₙ** can retrieve data without "waiting" for the peer to

25   complete a data-sending operation.

[74]   Similarly, the output DPSRAM **102** includes an input port **110** for receiving data from the hardwired pipelines **74₁-74ₙ** via the communication shell **84**, and includes an output port **112** for providing this data to a peer, such as the host processor **42** (**FIG. 3**), via the communication interface **82**. As discussed above, the

30   two data ports **110** (input) and **112** (output) increase the speed and efficiency of data transfer to/from the DPSRAM **102**, and using the DPSRAM **102** to buffer data from the pipelines **74₁-74ₙ** allows the peer and the pipelines to operate asynchronously

17

relative to one another. That is, the pipelines 74_1-74_n can publish data to the peer without "waiting" for the output-data handler *126* to complete a data transfer to the peer or to another peer. Likewise, the output-data handler *126* can transfer data to a peer without "waiting" for the pipelines 74_1-74_n to complete a data-publishing

5   operation.

[75]   The working DPSRAM *104* includes an input port *114* for receiving data from the hardwired pipelines 74_1-74_n via the communication shell *84*, and includes an output port *116* for returning this data back to the pipelines via the communication shell. While processing input data received from the DPSRAM *100*,

10   the pipelines 74_1-74_n may need to temporarily store partially processed, *i.e.*, intermediate, data before continuing the processing of this data. For example, a first pipeline, such as the pipeline 74_1, may generate intermediate data for further processing by a second pipeline, such as the pipeline 74_2; thus, the first pipeline may need to temporarily store the intermediate data until the second pipeline

15   retrieves it. The working DPSRAM *104* provides this temporary storage. As discussed above, the two data ports *114* (input) and *116* (output) increase the speed and efficiency of data transfer between the pipelines 74_1-74_n and the DPSRAM *104*. Furthermore, including a separate working DPSRAM *104* typically increases the speed and efficiency of the pipeline circuit *80* by allowing the DPSRAMs *100* and

20   *102* to function exclusively as data-input and data-output buffers, respectively. But, with slight modification to the pipeline circuit *80*, either or both of the DPSRAMS *100* and *102* can also be a working memory for the pipelines 74_1-74_n when the DPSRAM *104* is omitted, and even when it is present.

[76]   Although the DPSRAMS *100*, *102*, and *104* are described as being

25   external to the pipeline circuit *80*, one or more of these DPSRAMS, or equivalents thereto, may be internal to the pipeline circuit.

[77]   Still referring to **FIG. 5**, the communication interface *82* includes an industry-standard bus adapter *118*, an input-data handler *120*, input-data and input-event queues *122* and *124*, an output-data handler *126*, and output-data and

30   output-event queues *128* and *130*. Although the queues *122*, *124*, *128*, and *130* are shown as single queues, one or more of these queues may include sub queues (not

shown) that allow segregation by, e.g., priority, of the values stored in the queues or of the respective data that these values represent.

[78]   The industry-standard bus adapter *118* includes the physical layer that allows the transfer of data between the pipeline circuit *80* and the pipeline bus *50*

5   (FIG. 4) via the communication bus *94*. Therefore, if one wishes to change the parameters of the bus *94*, then he need only modify the adapter *118* and not the entire communication interface *82*. Where the industry-standard bus interface *91* is omitted from the pipeline unit *78*, , then the adapter *118* may be modified to allow the transfer of data directly between the pipeline bus *50* and the pipeline circuit *80*.

10   In this latter implementation, the modified adapter *118* includes the functionality of the bus interface *91*, and one need only modify the adapter *118* if he/she wishes to change the parameters of the bus *50*.

[79]   The input-data handler *120* receives data from the industry-standard adapter *118*, loads the data into the DPSRAM *100* via the input port *106*, and

15   generates and stores a pointer to the data and a corresponding data identifier in the input-data queue *122*. If the data is the payload of a message from a peer, such as the host processor *42* (FIG. 3), then the input-data handler *120* extracts the data from the message before loading the data into the DPSRAM *100*. The input-data handler *120* includes an interface *132*, which writes the data to the input port *106* of

20   the DPSRAM *100* and which is further discussed below in conjunction with **FIG. 6**. Alternatively, the input-data handler *120* can omit the extraction step and load the entire message into the DPSRAM *100*.

[80]   The input-data handler *120* also receives events from the industry-standard bus adapter *118*, and loads the events into the input-event

25   queue *124*.

[81]   Furthermore, the input-data handler *120* includes a validation manager *134*, which determines whether received data or events are intended for the pipeline circuit *80*. The validation manager *134* may make this determination by analyzing the header (or a portion thereof) of the message that contains the data or

30   the event, by analyzing the type of data or event, or the analyzing the instance identification (*i.e.*, the hardwired pipeline *74* for which the data/event is intended) of

19

the data or event. If the input-data handler **120** receives data or an event that is not intended for the pipeline circuit **80**, then the validation manager **134** prohibits the input-data handler from loading the received data/even. Where the peer-vector machine **40** includes the router **61** (**FIG. 3**) such that the pipeline unit **78** should

5   receive only data/events that are intended for the pipeline unit, the validation manager **134** may also cause the input-data handler **120** to send to the host processor **42** (**FIG. 3**) an exception message that identifies the exception (erroneously received data/event) and the peer that caused the exception.

[82]   The output-data handler **126** retrieves processed data from locations of

10 the DPSRAM **102** pointed to by the output-data queue **128**, and sends the processed data to one or more peers, such as the host processor **42** (**FIG. 3**), via the industry-standard bus adapter **118**. The output-data handler **126** includes an interface **136**, which reads the processed data from the DPSRAM **102** via the port **112**. The interface **136** is further discussed below in conjunction with **FIG. 7**.

15 [83]   The output-data handler **126** also retrieves from the output-event queue **130** events generated by the pipelines $74_1 - 74_n$, and sends the retrieved events to one or more peers, such as the host processor **42** (**FIG. 3**) via the industry-standard bus adapter **118**.

[84]   Furthermore, the output-data handler **126** includes a subscription

20 manager **138**, which includes a list of peers, such as the host processor **42** (**FIG. 3**), that subscribe to the processed data and to the events; the output-data handler uses this list to send the data/events to the correct peers. If a peer prefers the data/event to be the payload of a message, then the output-data handler **126** retrieves the network or bus-port address of the peer from the subscription manager **138**,

25 generates a header that includes the address, and generates the message from the data/event and the header.

[85]   Although the technique for storing and retrieving data stored in the DPSRAMS **100** and **102** involves the use of pointers and data identifiers, one may modify the input- and output-data handlers **120** and **126** to implement other

30 data-management techniques. Conventional examples of such data-management techniques include pointers using keys or tokens, input/output control (IOC) block, and spooling.

[86]   The communication shell *84* includes a physical layer that interfaces the hardwired pipelines *74₁-74ₙ* to the output-data queue *128*, the controller *86*, and the DPSRAMs *100*, *102*, and *104*. The shell *84* includes interfaces *140* and *142*, and optional interfaces *144* and *146*. The interfaces *140* and *146* may be similar to the interface *136*; the interface *140* reads input data from the DPSRAM *100* via the port *108*, and the interface *146* reads intermediate data from the DPSRAM *104* via the port *116*. The interfaces *142* and *144* may be similar to the interface *132*; the interface *142* writes processed data to the DPSRAM *102* via the port *110*, and the interface *144* writes intermediate data to the DPSRAM *104* via the port *114*.

[87]   The controller *86* includes a sequence manager *148* and a synchronization interface *150*, which receives one or more synchronization signals SYNC. A peer, such as the host processor *42* (FIG. 3), or a device (not shown) external to the peer-vector machine *40* (FIG. 3) may generate the SYNC signal, which triggers the sequence manager *148* to activate the hardwired pipelines *74₁-74ₙ* as discussed below and in previously cited U.S. Patent App. Serial No. 10/683,932 entitled PIPELINE ACCELERATOR HAVING MULTIPLE PIPELINE UNITS AND RELATED COMPUTING MACHINE AND METHOD. The synchronization interface *150* may also generate a SYNC signal to trigger the pipeline circuit *80* or to trigger another peer. In addition, the events from the input-event queue *124* also trigger the sequence manager *148* to activate the hardwired pipelines *74₁-74ₙ* as discussed below.

[88]   The sequence manager *148* sequences the hardwired pipelines *74₁-74ₙ* through their respective operations via the communication shell *84*. Typically, each pipeline *74* has at least three operating states: preprocessing, processing, and post processing. During preprocessing, the pipeline *74*, e.g., initializes its registers and retrieves input data from the DPSRAM *100*. During processing, the pipeline *74*, e.g., operates on the retrieved data, temporarily stores intermediate data in the DPSRAM *104*, retrieves the intermediate data from the DPSRAM *104*, and operates on the intermediate data to generate result data. During post processing, the pipeline *74*, e.g., loads the result data into the DPSRAM *102*. Therefore, the sequence manager *148* monitors the operation of the pipelines *74₁-74ₙ* and instructs each pipeline when to begin each of its operating states. And one may distribute the

pipeline tasks among the operating states differently than described above. For
example, the pipeline **74** may retrieve input data from the DPSRAM **100** during the
processing state instead of during the preprocessing state.

[89]   Furthermore, the sequence manager **148** maintains a predetermined
internal operating synchronization among the hardwired pipelines 74_1-74_n. For
example, to avoid all of the pipelines 74_1-74_n simultaneously retrieving data from the
DPSRAM **100**, it may be desired to synchronize the pipelines such that while the first
pipeline 74_1 is in a preprocessing state, the second pipeline 74_2 is in a processing
state and the third pipeline 74_3 is in a post-processing state. Because a state of one
pipeline **74** may require a different number of clock cycles than a concurrently
performed state of another pipeline, the pipelines 74_1-74_n may lose synchronization if
allowed to run freely. Consequently, at certain times there may be a "bottle neck,"
as, for example, multiple pipelines **74** simultaneously attempt to retrieve data from
the DPSRAM **100**. To prevent the loss of synchronization and its undesirable
consequences, the sequence manager **148** allows all of the pipelines **74** to complete
a current operating state before allowing any of the pipelines to proceed to a next
operating state. Therefore, the time that the sequence manager **148** allots for a
current operating state is long enough to allow the slowest pipeline **74** to complete
that state. Alternatively, circuitry (not shown) for maintaining a predetermined
operating synchronization among the hardwired pipelines 74_1-74_n may be included
within the pipelines themselves.

[90]   In addition to sequencing and internally synchronizing the hardwired
pipelines 74_1 - 74_n, the sequence manager **148** synchronizes the operation of the
pipelines to the operation of other peers, such as the host processor **42** (**FIG. 3**), and
to the operation of other external devices in response to one or more SYNC signals
or to an event in the input-events queue **124**.

[91]   Typically, a SYNC signal triggers a time-critical function but requires
significant hardware resources; comparatively, an event typically triggers a
non-time-critical function but requires significantly fewer hardware resources. As
discussed in previously cited U.S. Patent App. Serial No. 10/683,932 entitled
PIPELINE ACCELERATOR HAVING MULTIPLE PIPELINE UNITS AND RELATED
COMPUTING MACHINE AND METHOD, because a SYNC signal is routed directly

from peer to peer, it can trigger a function more quickly than an event, which must makes its way through, e.g., the pipeline bus **50** (FIG. 3), the input-data handler **120**, and the input-event queue **124**. But because they are separately routed, the SYNC signals require dedicated circuitry, such as routing lines, buffers, and the SYNC

5   interface **150**, of the pipeline circuit **80**. Conversely, because they use the existing data-transfer infrastructure (e.g. the pipeline bus **50** and the input-data handler **120**), the events require only the dedicated input-event queue **124**. Consequently, designers tend to use events to trigger all but the most time-critical functions.

[92]   The following is an example of function triggering. Assume that a

10   sonar sensor element (not shown) sends blocks of data to the pipeline unit **78**, the input-data handler **120** stores this data in the DPSRAM **100**, the pipeline **74₁** transfers this data from the DPSRAM **100** to the DPSRAM **104**, and, when triggered, the pipeline **74₂** retrieves and processes the data from the DPSRAM **104**. If the processing that the pipeline **74₂** performs on the data is time critical, then the sensor

15   element may generate a SYNC pulse to trigger the pipeline **74₂**, via the interface **150** and the sequence manager **148**, as soon as the pipeline **74₁** finishes loading an entire block of data into the DPSRAM **104**. There are many conventional techniques that the pipeline unit **78** and the sensor can employ to determine when the pipeline **74₁** is finished. For example, as discussed below, the sequence manager **148** may

20   provide a corresponding SYNC pulse or event to the sensor. Alternatively, if the processing that the pipeline **74₂** performs is not time critical, then the sensor may send an event to the sequence manager **148** via the pipeline bus **50** (FIG. 3).

[93]   The sequence manager **148** may also provide to a peer, such as the host processor **42** (FIG. 3), information regarding the operation of the hardwired

25   pipelines **74₁-74ₙ** by generating a SYNC pulse or an event. The sequence manager **148** sends a SYNC pulse via the SYNC interface **150** and a dedicated line (not shown), and sends an event via the output-event queue **130** and the output-data handler **126**. Referring to the above example, suppose that a peer further processes the data blocks from the pipeline **74₂**. The sequence manager **148** may notify the

30   peer via a SYNC pulse or an event when the pipeline **74₂** has finished processing a block of data. The sequence manager **148** may also confirm receipt of a SYNC

pulse or an event by generating and sending a corresponding SYNC pulse or event to the appropriate peer(s).

[94]   Still referring to **FIG. 5**, the operation of the pipeline unit **78** is discussed according to an embodiment of the invention.

5   [95]   For data, the industry-standard bus interface **91** receives data signals (which originates from a peer, such as the host processor **42** of **FIG. 3**) from the pipeline bus **50** (and the router **61** if present), and translates these signals into messages each having a header and payload.

[96]   Next, the industry-standard bus adapter **118** converts the messages

10 from the industry-standard bus interface **91** into a format that is compatible with the input-data handler **120**.

[97]   Then, the input-data handler **120** dissects the message headers and extracts from each header the portion that describes the data payload. For example, the extracted header portion may include, e.g., the address of the pipeline unit **78**,

15 the type of data in the payload, or an instance identifier that identifies the pipeline(s) **78₁ – 78ₙ** for which the data is intended.

[98]   Next, the validation manager **134** analyzes the extracted header portion and confirms that the data is intended for one of the hardwired pipelines **74₁**-**74ₙ**, the interface **132** writes the data to a location of the DPSRAM **100** via the port

20 **106**, and the input-data handler **120** stores a pointer to the location and a corresponding data identifier in the input-data queue **122**. The data identifier identifies the pipeline or pipelines **74₁-74ₙ** for which the data is intended, or includes information that allows the sequence manager **148** to make this identification as discussed below. Alternatively, the queue **122** may include a respective subqueue

25 (not shown) for each pipeline **74₁-74ₙ**, and the input-data handler **120** stores the pointer in the subqueue or subqueues of the intended pipeline or pipelines. In this alternative, the data identifier may be omitted. Furthermore, if the data is the payload of a message, then the input-data handler **120** extracts the data from the message before the interface **132** stores the data in the DPSRAM **100**.

30 Alternatively, as discussed above, the interface **132** may store the entire message in the DPSRAM **100**.

[99]   Then, at the appropriate time, the sequence manager *148* reads the pointer and the data identifier from the input-data queue *122*, determines from the data identifier the pipeline or pipelines *74₁ – 74ₙ* for which the data is intended, and passes the pointer to the pipeline or pipelines via the communication shell *84*.

5   [100]   Next, the data-receiving pipeline or pipelines *74₁ – 74ₙ* cause the interface *140* to retrieve the data from the pointed-to location of the DPSRAM *100* via the port *108*.

[101]   Then, the data-receiving pipeline or pipelines *74₁-74ₙ* process the retrieved data, the interface *142* writes the processed data to a location of the

10   DPSRAM *102* via the port *110*, and the communication shell *84* loads into the output-data queue *128* a pointer to and a data identifier for the processed data. The data identifier identifies the destination peer or peers, such as the host processor *42* (FIG. 3), that subscribe to the processed data, or includes information (such as the data type) that allows the subscription manager *138* to subsequently determine the

15   destination peer or peers (*e.g.*, the host processor *42* of FIG. 3). Alternatively, the queue *128* may include a respective subqueue (not shown) for each pipeline *74₁-74ₙ*, and the communication shell *84* stores the pointer in the subqueue or subqueues of the originating pipeline or pipelines. In this alternative, the communication shell *84* may omit loading a data identifier into the queue *128*.

20   Furthermore, if the pipeline or pipelines *74₁-74ₙ* generate intermediate data while processing the retrieved data, then the interface *144* writes the intermediate data into the DPSRAM *104* via the port *114*, and the interface *146* retrieves the intermediate data from the DPSRAM *104* via the port *116*.

[102]   Next, the output-data handler *126* retrieves the pointer and the data

25   identifier from the output-data queue *128*, the subscription manager *138* determines from the identifier the destination peer or peers (*e.g.*, the host processor *42* of FIG. 3) of the data, the interface *136* retrieves the data from the pointed-to location of the DPSRAM *102* via the port *112*, and the output-data handler sends the data to the industry-standard bus adapter *118*. If a destination peer requires the data to be the

30   payload of a message, then the output-data handler *126* generates the message and sends the message to the adapter *118*. For example, suppose the data has multiple destination peers and the pipeline bus *50* supports message broadcasting.

25

The output-data handler **126** generates a single header that includes the addresses of all the destination peers, combines the header and data into a message, and sends (via the adapter **118** and the industry-standard bus interface **91**) a single message to all of the destination peers simultaneously. Alternatively, the output-

5   data handler **126** generates a respective header, and thus a respective message, for each destination peer, and sends each of the messages separately.

[103]   Then, the industry-standard bus adapter **118** formats the data from the output-data handler **126** so that it is compatible with the industry-standard bus interface **91**.

10   [104]   Next, the industry-standard bus interface **91** formats the data from the industry-standard bus adapter **118** so that it is compatible with the pipeline bus **50** (FIG. 3).

[105]   For an event with no accompanying data, *i.e.*, a doorbell, the industry-standard bus interface **91** receives a signal (which originates from a peer,

15   such as the host processor **42** of **FIG. 3**) from the pipeline bus **50** (and the router **61** if present), and translates the signal into a header (*i.e.*, a data-less message) that includes the event.

[106]   Next, the industry-standard bus adapter **118** converts the header from the industry-standard bus interface **91** into a format that is compatible with the

20   input-data handler **120**.

[107]   Then, the input-data handler **120** extracts from the header the event and a description of the event. For example, the description may include, *e.g.*, the address of the pipeline unit **78**, the type of event, or an instance identifier that identifies the pipeline(s) $78_1 - 78_n$ for which the event is intended.

25   [108]   Next, the validation manager **134** analyzes the event description and confirms that the event is intended for one of the hardwired pipelines 74_1-74_n, and the input-data handler **120** stores the event and its description in the input-event queue **124**.

[109]   Then, at the appropriate time, the sequence manager **148** reads the

30   event and its description from the input-event queue **124**, and, in response to the event, triggers the operation of one or more of the pipelines 74_1-74_n as discussed

above. For example, the sequence manager *148* may trigger the pipeline *74₂* to begin processing data that the pipeline *74₁* previously stored in the DPSRAM *104*.

[110]   To output an event, the sequence manager *148* generates the event and a description of the event, and loads the event and its description into the output-event queue *130* — the event description identifies the destination peer(s) for the event if there is more than one possible destination peer. For example, as discussed above, the event may confirm the receipt and implementation of an input event, an input-data or input-event message, or a SYNC pulse

[111]   Next, the output-data handler *126* retrieves the event and its description from the output-event queue *130*, the subscription manager *138* determines from the event description the destination peer or peers (e.g., the host processor *42* of **FIG. 3**) of the event, and the output-data handler sends the event to the proper destination peer or peers via the industry-standard bus adapter *118* and the industry-standard bus interface *91* as discussed above.

[112]   For a configuration command, the industry-standard bus adapter *118* receives the command from the host processor *42* (**FIG. 3**) via the industry-standard bus interface *91*, and provides the command to the input-data handler *120* in a manner similar to that discussed above for a data-less event (*i.e.*, doorbell)

[113]   Next, the validation manager *134* confirms that the command is intended for the pipeline unit *78*, and the input-data handler *120* loads the command into the configuration manager *90*. Furthermore, either the input-data handler *120* or the configuration manager *90* may also pass the command to the output-data handler *126*, which confirms that the pipeline unit *78* received the command by sending the command back to the peer (e.g., the host processor *42* of **FIG. 3**) that sent the command. This confirmation technique is sometimes called "echoing."

[114]   Then, the configuration manager *90* implements the command. For example, the command may cause the configuration manager *90* to disable one of the pipelines *74₁-74ₙ* for debugging purposes. Or, the command may allow a peer, such as the host processor *42* (**FIG. 3**), to read the current configuration of the pipeline circuit *80* from the configuration manager *90* via the output-data handler

27

126. In addition, one may use a configuration command to define an exception that is recognized by the exception manager *88*.

[115]   For an exception, a component, such as the input-data queue *122*, of the pipeline circuit *80* triggers an exception to the exception manager *88*. In one
5   implementation, the component includes an exception-triggering adapter (not shown) that monitors the component and triggers the exception in response to a predetermined condition or set of conditions. The exception-triggering adapter may be a universal circuit that can be designed once and then included as part of each component of the pipeline circuit *80* that generates exceptions.

10   [116]   Next, in response to the exception trigger, the exception manager *88* generates an exception identifier. For example, the identifier may indicate that the input-data queue *122* has overflowed. Furthermore, the identifier may include its destination peer if there is more than one possible destination peer.

[117]   Then, the output-data handler *126* retrieves the exception identifier
15   from the exception manager *88* and sends the exception identifier to the host processor *42* (FIG. 3) as discussed in previously cited U.S. Patent App. Serial No. 10/684,053 entitled COMPUTING MACHINE HAVING IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD. Alternatively, if there are multiple possible destination peers, then the exception identifier can also include
20   destination information from which the subscription manager *138* determines the destination peer or peers (e.g., the host processor *42* of FIG. 3) of the identifier. The output-data handler *126* then sends the identifier to the destination peer or peers via the industry-standard bus adapter *118* and the industry-standard bus interface *91*.

[118]   Still referring to FIG. 5, alternative embodiments to the pipeline unit *78*
25   exist. For example, although described as including DPSRAMs, the data memory *92* may include other types of memory ICs such as quad-data-rate (QDR) SRAMs.

[119]   FIG. 6 is a block diagram of the interface *142* of FIG. 5 according to an embodiment of the invention. As discussed above in conjunction with FIG. 5, the interface *142* writes processed data from the hardwired pipelines *74₁-74ₙ* to the
30   DPSRAM *102*. As discussed below, the structure of the interface *142* reduces or

eliminates data "bottlenecks" and, where the pipeline circuit **80** (FIG. 5) is a PLIC, makes efficient use of the PLIC's local and global routing resources.

[120]   The interface **142** includes write channels $150_1 - 150_n$, one channel for each hardwired pipeline $74_1 - 74_n$ (FIG. 5), and includes a controller **152**. For

5   purposes of illustration, the channel 150_1 is discussed below, it being understood that the operation and structure of the other channels $150_2 - 150_n$ are similar unless stated otherwise.

[121]   The channel 150_1 includes a write-address/data FIFO 154_1 and a address/data register 156_1.

10   [122]   The FIFO 154_1 stores the data that the pipeline 74_1 writes to the DPSRAM **102**, and stores the address of the location within the DPSRAM **102** to which the pipeline writes the data, until the controller **152** can actually write the data to the DPSRAM **102** via the register 156_1. Therefore, the FIFO 154_1 reduces or eliminates the data bottleneck that may occur if the pipeline 74_1 had to "wait" to write

15   data to the channel 150_1 until the controller **152** finished writing previous data.

[123]   The FIFO 154_1 receives the data from the pipeline 74_1 via a bus 158_1, receives the address of the location to which the data is to be written via a bus 160_1, and provides the data and address to the register 156_1 via busses 162_1 and 164_1, respectively. Furthermore, the FIFO 154_1 receives a WRITE FIFO signal from the

20   pipeline 74_1 on a line 166_1, receives a CLOCK signal via a line 168_1, and provides a FIFO FULL signal to the pipeline 74_1 on a line 170_1. In addition, the FIFO 154_1 receives a READ FIFO signal from the controller **152** via a line 172_1, and provides a FIFO EMPTY signal to the controller via a line 174_1. Where the pipeline circuit **80** (FIG. 5) is a PLIC, the busses 158_1, 160_1, 162_1, and 164_1 and the lines 166_1, 168_1,

25   170_1, 172_1, and 174_1 are preferably formed using local routing resources. Typically, local routing resources are preferred to global routing resources because the signal-path lengths are generally shorter and the routing is easier to implement.

[124]   The register 156_1 receives the data to be written and the address of the write location from the FIFO 154_1 via the busses 162_1 and 164_1, respectively,

30   and provides the data and address to the port **110** of the DPSRAM **102** (FIG. 5) via an address/data bus **176**. Furthermore, the register 156_1 also receives the data and

address from the registers $156_2 - 156_n$ via an address/data bus 178_1 as discussed below. In addition, the register 156_1 receives a SHIFT/LOAD signal from the controller 152 via a line 180. Where the pipeline circuit 80 (FIG. 5) is a PLIC, the bus 176 is typically formed using global routing resources, and the busses $178_1 -$

5   178_{n-1} and the line 180 are preferably formed using local routing resources.

[125]   In addition to receiving the FIFO EMPTY signal and generating the READ FIFO and SHIFT/LOAD signals, the controller 152 provides a WRITE DPSRAM signal to the port 110 of the DPSRAM 102 (FIG. 5) via a line 182.

[126]   Still referring to FIG. 6, the operation of the interface 142 is discussed.

10   [127]   First, the FIFO 154_1 drives the FIFO FULL signal to the logic level corresponding to the current state ("full" or "not full") of the FIFO.

[128]   Next, if the FIFO 154_1 is not full and the pipeline 74_1 has processed data to write, the pipeline drives the data and corresponding address onto the busses 158_1 and 160_1, respectively, and asserts the WRITE signal, thus loading the

15   data and address into the FIFO. If the FIFO 154_1 is full, however, the pipeline 74_1 waits until the FIFO is not full before loading the data.

[129]   Then, the FIFO 154_1 drives the FIFO EMPTY signal to the logic level corresponding to the current state ("empty" or "not empty") of the FIFO.

[130]   Next, if the FIFO 154_1 is not empty, the controller 152 asserts the

20   READ FIFO signal and drives the SHIFT/LOAD signal to the load logic level, thus loading the first loaded data and address from the FIFO into the register 156_1. If the FIFO 154_1 is empty, the controller 152 does not assert READ FIFO, but does drive SHIFT load to the load logic level if any of the other FIFOs 154_2-154_n are not empty.

[131]   The channels $150_2 - 150_n$ operate in a similar manner such that first-

25   loaded data in the FIFOs $154_2 - 154_n$ are respectively loaded into the registers $156_2.156_n$.

[132]   Then, the controller 152 drives the SHIFT/LOAD signal to the shift logic level and asserts the WRITE DPSRAM signal, thus serially shifting the data and addresses from the registers $156_1 - 156_n$ onto the address/data bus 176 and loading

30   the data into the corresponding locations of the DPSRAM 102. Specifically, during a

first shift cycle, the data and address from the register 156_1 are shifted onto the bus
176 such that the data from the FIFO 154_1 is loaded into the addressed location of
the DPSRAM **102**. Also during the first shift cycle, the data and address from the
register 156_2 are shifted into the register 156_1, the data and address from the

5   register 156_3 (not shown) are shifted into the register 156_2, and so on. During a
second shift cycle, the data and address from the register 156_1 are shifted onto the
bus **176** such that the data from the FIFO 154_2 is loaded into the addressed location
of the DPSRAM **102**. Also during the second shift cycle, the data and address from
the register 156_2 are shifted into the register 156_1, the data and address from the

10   register 156_3 (not shown) are shifted into the register 156_2, and so on. There are n
shift cycles, and during the nth shift cycle the data and address from the register
156_n (which is the data and address from the FIFO 154_n) is shifted onto the bus **176**.
The controller **152** may implement these shift cycles by pulsing the SHIFT/LOAD
signal, or by generating a shift clock signal (not shown) that is coupled to the

15   registers 156_1-156_n. Furthermore, if one of the registers 156_1-156_n is empty during a
particular shift operation because its corresponding FIFO 154_1-154_n was empty
when the controller **152** loaded the register, then the controller may bypass the
empty register, and thus shorten the shift operation by avoiding shifting null data and
a null address onto the bus **176**.

20   [133]   Referring to **FIGS. 5** and **6**, according to an embodiment of the
invention, the interface **144** is similar to the interface **142**, and the interface **132** is
also similar to the interface **142** except that the interface **132** includes only one write
channel **150**.

[134]   **FIG. 7** is a block diagram of the interface **140** of **FIG. 5** according to an
25   embodiment of the invention. As discussed above in conjunction with **FIG. 5**, the
interface **140** reads input data from the DPSRAM **100** and transfers this data to the
hardwired 74_1-74_n. As discussed below, the structure of the interface **140** reduces
or eliminates data "bottlenecks" and, where the pipeline circuit **80** (**FIG. 5**) is a PLIC,
makes efficient use of the PLIC's local and global routing resources.

30   [135]   The interface **140** includes read channels 190_1 – 190_n, one channel for
each hardwired pipeline 74_1 – 74_n (**FIG. 5**), and a controller **192**. For purposes of
illustration, the read channel 190_1 is discussed below, it being understood that the

31

operation and structure of the other read channels $190_2 - 190_n$ are similar unless stated otherwise.

[136]   The channel 190_1 includes a FIFO 194_1 and an address/identifier (ID) register 196_1. As discussed below, the identifier identifies the pipeline 74_1-74_n that
5   makes the request to read data from a particular location of the DPSRAM *100* to receive the data.

[137]   The FIFO 194_1 includes two sub-FIFOs (not shown), one for storing the address of the location within the DPSRAM *100* from which the pipeline 74_1 wishes to read the input data, and the other for storing the data read from the DPSRAM
10   *100*. Therefore, the FIFO 194_1 reduces or eliminates the bottleneck that may occur if the pipeline 74_1 had to "wait" to provide the read address to the channel 190_1 until the controller *192* finished reading previous data, or if the controller had to wait until the pipeline 74_1 retrieved the read data before the controller could read subsequent data.

15   [138]   The FIFO 194_1 receives the read address from the pipeline 74_1 via a bus 198_1 and provides the address and ID to the register 196_1 via a bus 200_1. Since the ID corresponds to the pipeline 74_1 and typically does not change, the FIFO 194_1 may store the ID and concatenate the ID with the address. Alternatively, the pipeline 74_1 may provide the ID to the FIFO 194_1 via the bus 198_1. Furthermore, the FIFO
20   194_1 receives a READY WRITE FIFO signal from the pipeline 74_1 via a line 202_1, receives a CLOCK signal via a line 204_1, and provides a FIFO FULL (of read addresses) signal to the pipeline via a line 206_1. In addition, the FIFO 194_1 receives a WRITE/READ FIFO signal from the controller *192* via a line 208_1, and provides a FIFO EMPTY signal to the controller via a line 210_1. Moreover, the FIFO 194_1
25   receives the read data and the corresponding ID from the controller *192* via a bus *212*, and provides this data to the pipeline 74_1 via a bus 214_1. Where the pipeline circuit *80* (FIG. 5) is a PLIC, the busses 198_1, 200_1, and 214_1 and the lines 202_1, 204_1, 206_1, 208_1, and 210_1 are preferably formed using local routing resources, and the bus *212* is typically formed using global routing resources.

30   [139]   The register 196_1 receives the address of the location to be read and the corresponding ID from the FIFO 194_1 via the bus 206_1, provides the address to the port *108* of the DPSRAM *100* (FIG. 5) via an address bus *216*, and provides the

ID to the controller **192** via a bus **218**. Furthermore, the register **196₁** also receives the addresses and IDs from the registers **196₂ – 196ₙ** via an address/ID bus **220₁** as discussed below. In addition, the register **196₁** receives a SHIFT/LOAD signal from the controller **192** via a line **222**. Where the pipeline circuit **80** (FIG. 5) is a PLIC, the

5　bus **216** is typically formed using global routing resources, and the busses **220₁-220ₙ₋₁** and the line **222** are preferably formed using local routing resources.

[140]　In addition to receiving the FIFO EMPTY signal, generating the WRITE/READ FIFO and SHIFT/LOAD signals, and providing the read data and corresponding ID, the controller **192** receives the data read from the port **108** of the

10　DPSRAM **100** (FIG. 5) via a bus **224** and generates a READ DPSRAM signal on a line **226**, which couples this signal to the port **108**. Where the pipeline circuit **80** (FIG. 5) is a PLIC, the bus **224** and the line **226** are typically formed using global routing resources.

[141]　Still referring to **FIG. 7**, the operation of the interface **140** is discussed.

15　[142]　First, the FIFO **194₁** drives the FIFO FULL signal to the logic level corresponding to the current state ("full" or "not full") of the FIFO relative to the read addresses. That is, if the FIFO **194₁** is full of addresses to be read, then it drives the logic level of FIFO FULL to one level, and if the FIFO is not full of read addresses, it drives the logic level of FIFO FULL to another level.

20　[143]　Next, if the FIFO **194₁** is not full of read addresses and the pipeline **74₁** is ready for more input data to process, the pipeline drives the address of the data to be read onto the bus **198₁**, and asserts the READ/WRITE FIFO signal to a write level, thus loading the address into the FIFO. As discussed above in conjunction with **FIG. 5**, the pipeline **74₁** gets the address from the input-data queue **122** via the

25　sequence manager **148**. If, however, the FIFO **194₁** is full of read addresses, the pipeline **74₁** waits until the FIFO is not full before loading the read address.

[144]　Then, the FIFO **194₁** drives the FIFO EMPTY signal to the logic level corresponding to the current state ("empty" or "not empty") of the FIFO relative to the read addresses. That is, if the FIFO **194₁** is loaded with at least one read address, it

30　drives the logic level of FIFO EMPTY to one level, and if the FIFO is loaded with no read addresses, it drives the logic level of FIFO EMPTY to another level.

[145] Next, if the FIFO 194_1 is not empty, the controller 192 asserts the WRITE/READ FIFO signal to the read logic level and drives the SHIFT/LOAD signal to the load logic level, thus loading the first loaded address and the ID from the FIFO into the register 196_1.

5 **[146]** The channels $190_2 - 190_n$ operate in a similar manner such that the controller 192 respectively loads the first-loaded addresses and IDs from the FIFOs $194_2 - 194_n$ into the registers $196_2 - 196_n$. If all of the FIFOs 194_2-194_n are empty, then the controller 192 waits for at least one of the FIFOs to receive an address before proceeding.

10 **[147]** Then, the controller 192 drives the SHIFT/LOAD signal to the shift logic level and asserts the READ DPSRAM signal to serially shift the addresses and IDs from the registers $196_1 - 196_n$ onto the address and ID busses 216 and 218 and to serially read the data from the corresponding locations of the DPSRAM 100 via the bus 224.

15 **[148]** Next, the controller 192 drives the received data and corresponding ID — the ID allows each of the FIFOs $194_1 - 194_n$ to determine whether it is an intended recipient of the data — onto the bus 212, and drives the WRITE/READ FIFO signal to a write level, thus serially writing the data to the respective FIFO, 194_1-194_n.

20 **[149]** Then, the hardwired pipelines 74_1-74_n sequentially assert their READ/WRITE FIFO signals to a read level and sequentially read the data via the busses 214_1-214_n.

[150] Still referring to **FIG. 7**, a more detailed discussion of their data-read operator is presented.

25 **[151]** During a first shift cycle, the controller 192 shifts the address and ID from the register 196_1 onto the busses 216 and 218, respectively, asserts read DPSRAM, and thus reads the data from the corresponding location of the DPSRAM 100 via the bus 224 and reads the ID from the bus 218. Next, the controller 192 drives WRITE/READ FIFO signal on the line 208_1 to a write level and drives the

30 received data and the ID onto the bus 212. Because the ID is the ID from the FIFO 194_1, the FIFO 194_1 recognizes the ID and thus loads the data from the bus 212 in

34

response the write level of the WRITE/READ FIFO signal. The remaining FIFOs $194_2 - 194_n$ do not load the data because the ID on the bus 212 does not correspond to their IDs. Then, the pipeline 74_1 asserts the READ/WRITE FIFO signal on the line 202_1 to the read level and retrieves the read data via the bus 214_1.

5   Also during the first shift cycle, the address and ID from the register 196_2 are shifted into the register 196_1, the address and ID from the register 196_3 (not shown) are shifted into the register 196_2, and so on. Alternatively, the controller 192 may recognize the ID and drive only the WRITE/READ FIFO signal on the line 208_1 to the write level. This eliminates the need for the controller 192 to send the ID to the

10   FIFOs 194_1-194_n. In another alternative, the WRITE/READ FIFO signal may be only a read signal, and the FIFO 194_1 (as well as the other FIFOs 194_2-194_n) may load the data on the bus 212 when the ID on the bus 212 matches the ID of the FIFO 194_1. This eliminates the need of the controller 192 to generate a write signal.

       [152]   During a second shift cycle, the address and ID from the register 196_1

15   is shifted onto the busses 216 and 218 such that the controller 192 reads data from the location of the DPSRAM 100 specified by the FIFO 194_2. Next, the controller 192 drives the WRITE/READ FIFO signal to a write level and drives the received data and the ID onto the bus 212. Because the ID is the ID from the FIFO 194_2, the FIFO 194_2 recognizes the ID and thus loads the data from the bus 212. The

20   remaining FIFOs 194_1 and $194_3 - 194_n$ do not load the data because the ID on the bus 212 does not correspond to their IDs. Then, the pipeline 74_2 asserts its READ/WRITE FIFO signal to the read level and retrieves the read data via the bus 214_2. Also during the second shift cycle, the address and ID from the register 196_2 is shifted into the register 196_1, the address and ID from the register 196_3 (not

25   shown) is shifted into the register 196_2, and so on.

       [153]   This continues for n shift cycles, i.e., until the address and ID from the register 196_n (which is the address and ID from the FIFO 194_n) are respectively shifted onto the bus 216 and 218. The controller 192 may implement these shift cycles by pulsing the SHIFT/LOAD signal, or by generating a shift clock signal (not

30   shown) that is coupled to the registers 196_1-196_n. Furthermore, if one of the registers 196_1-196_2 is empty during a particular shift operation because its corresponding FIFO 194_1-194_n is empty, then the controller 192 may bypass the

empty register, and thus shorten the shift operation by avoiding shifting a null address onto the bus **216**.

[154]   Referring to **FIGS. 5** and **6**, according to an embodiment of the invention, the interface **144** is similar to the interface **140**, and the interface **136** is

5   also similar to the interface **140** except that the interface **136** includes only one read channel **190**, and thus includes no ID circuitry.

[155]   **Fig. 8** is a schematic block diagram of a pipeline unit **230** of **FIG. 4** according to another embodiment of the invention. The pipeline unit **230** is similar to the pipeline unit **78** of **FIG. 4** except that the pipeline unit **230** includes multiple

10   pipeline circuits **80** — here two pipeline circuits **80a** and **80b**. Increasing the number of pipeline circuits **80** typically allows an increase in the number n of hardwired pipelines 74_1-74_n, and thus an increase in the functionality of the pipeline unit **230** as compared to the pipeline unit **78**.

[156]   In the pipeline unit **230** of **FIG. 8**, the services components, i.e., the

15   communication interface **82**, the controller **86**, the exception manager **88**, the configuration manager **90**, and the optional industry-standard bus interface **91**, are disposed on the pipeline circuit **80a**, and the pipelines 74_1-74_n and the communication shell **84** are disposed on the pipeline circuit **80b**. By locating the services components and the pipelines 74_1-74_n on separate pipeline circuits, one

20   can include a higher number n of pipelines and/or more complex pipelines than he can where the service components and the pipelines are located on the same pipeline circuit. Alternatively, the portion of the communication shell **84** that interfaces the pipelines 74_1-74_n to the interface **82** and the controller **86** may be disposed on the pipeline circuit **80a**.

25   [157]   **FIG. 9** is a schematic block diagram of the pipeline circuits **80a** and **80b** and the data memory **92** of the pipeline unit **230** of **FIG. 8** according to an embodiment of the invention. Other than the pipeline components being disposed on two pipelines circuits, the structure and operation of the pipeline circuits **80a** and **80b** and the memory **92** of **FIG. 9** are the same as for the pipeline circuit **80** and

30   memory **92** of **FIG. 5**.

[158]   The preceding discussion is presented to enable a person skilled in the art to make and use the invention. Various modifications to the embodiments will be readily apparent to those skilled in the art, and the generic principles herein may be applied to other embodiments and applications without departing from the spirit and scope of the present invention. Thus, the present invention is not intended to be limited to the embodiments shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

WHAT IS CLAIMED IS:

1.   A pipeline accelerator, comprising:

a memory; and

a hardwired-pipeline circuit coupled to the memory and operable to,

        receive data,

        load the data into the memory,

        retrieve the data from the memory,

        process the retrieved data, and

        provide the processed data to an external source.

2.   The pipeline accelerator of claim 1 wherein:

the memory is disposed on a first integrated circuit; and

the pipeline circuit is disposed on a second integrated circuit.

3.   The pipeline accelerator of claim 1 wherein the pipeline circuit is disposed on a field-programmable gate array.

4.   The pipeline accelerator of claim 1 wherein the pipeline circuit is operable to provide the processed data to the external source by:

loading the processed data into the memory;

retrieving the processed data from the memory; and

providing the retrieved processed data to the external source.

5.   The pipeline accelerator of claim 1 wherein:

the external source comprises a processor; and

the pipeline circuit is operable to receive the data from the processor.

6.   A computing machine, comprising:

a processor; and

a pipeline accelerator coupled to the processor and comprising,

        a memory, and

        a hardwired-pipeline circuit coupled to the memory and operable to,

            receive data from the processor,

            load the data into the memory,

            retrieve the data from the memory,

            process the retrieved data, and

provide the processed data to the processor.

7.   A pipeline accelerator, comprising:

a memory; and

a hardwired-pipeline circuit coupled to the memory and operable to,

  receive data,

  process the received data,

  load the processed data into the memory,

  retrieve the processed data from the memory, and

  provide the retrieved processed data to an external source.

8.   A computing machine, comprising:

a processor; and

a pipeline accelerator coupled to the processor and comprising,

  a memory, and

  a hardwired-pipeline circuit coupled to the memory and operable to,

    receive data from the processor,

    process the received data,

    load the processed data into the memory,

    retrieve the processed data from the memory, and

    provide the retrieved processed data to the processor.

9.   A pipeline accelerator, comprising:

first and second memories; and

a hardwired-pipeline circuit coupled to the first and second memories and comprising,

  an input-data handler operable to receive raw data from an external source and to load the raw data into the first memory,

  a hardwired pipeline operable to process the raw data,

  a pipeline interface operable to retrieve the raw data from the first memory, provide the retrieved raw data to the hardwired pipeline, and load processed data from the hardwired pipeline into the second memory, and

  an output-data handler operable to retrieve the processed data from the second memory and to provide the processed data to the external source.

10.   The pipeline accelerator of claim 9 wherein:

the first and second memories each include respective first and second ports;

the input-data handler is operable to load the raw data via the first port of the first memory,

the pipeline interface is operable to retrieve the raw data via the second port of the first memory and to load the processed data via the first port of the second memory, and

the output-data handler is operable to retrieve the processed data via the second port of the second memory.

11.   The pipeline accelerator of claim 9, further comprising:

a third memory coupled to the hardwired-pipeline circuit;

wherein the hardwired pipeline is operable to generate intermediate data while processing the raw data; and

wherein the pipeline interface is operable to load the intermediate data into the third memory and to retrieve the intermediate data from the third memory.

12.   The pipeline accelerator of claim 9 wherein:

the first and second memories are respectively disposed on first and second integrated circuits; and

the pipeline circuit is disposed on a field-programmable gate array.

13.   The pipeline accelerator of claim 9, further comprising:

an input-data queue coupled to the input-data handler and the pipeline interface;

wherein the input-data handler is operable to load into the input-data queue a pointer to a location of the raw data within the first memory; and

wherein the pipeline interface is operable to retrieve the raw data from the location using the pointer.

14.   The pipeline accelerator of claim 9, further comprising:

an output-data queue coupled to the output-data handler and the pipeline interface;

wherein the pipeline interface is operable to load into the output-data queue a pointer to a location of the processed data within the second memory; and

wherein the output-data handler is operable to retrieve the processed data from the location using the pointer.

15.   The pipeline accelerator of claim 9, further comprising:

wherein each of the input-data handler, hardwired pipeline, pipeline interface, and output-data handler has a respective operating configuration; and

a configuration manager coupled to and operable to set the operating configurations of the input-data handler, hardwired pipeline, pipeline interface, and output-data handler.

16.   The pipeline accelerator of claim 9, further comprising:

wherein each of the input-data handler, hardwired pipeline, pipeline interface, and output-data handler has a respective operating status; and

an exception manager coupled to and operable to identify an exception in the input-data handler, hardwired pipeline, pipeline interface, or output-data handler in response to the operating statuses.

17.   A pipeline accelerator, comprising:

a hardwired pipeline operable to process data; and

an input-data handler coupled to the hardwired pipeline and operable to,

    receive the data,

    determine whether the data is directed to the hardwired pipeline, and

    provide the data to the hardwired pipeline if the data is directed to the hardwired pipeline.

18.   The pipeline accelerator of claim 17 wherein the input-data handler is further operable to:

receive the data by,

    receiving a message that includes a header and the data, and

    extracting the data from the message; and

determine whether the data is directed to the hardwired pipeline by analyzing the header.

19.   The pipeline accelerator of claim 17 wherein the hardwired pipeline and the input-data handler are disposed on a single field-programmable gate array.

20.   The pipeline accelerator of claim 17 wherein the hardwired pipeline and the input-data handler are disposed on respective field-programmable gate arrays.

21.   A computing machine, comprising:

a processor; and

a pipeline accelerator coupled to the processor and comprising,

       a hardwired pipeline operable to process data, and

       an input-data handler coupled to the hardwired pipeline and operable

to,

              receive the data from the processor,

              determine whether the data is directed to the hardwired pipeline,

and

              provide the data to the hardwired pipeline if the data is directed

to the hardwired pipeline.

22.   A pipeline accelerator, comprising:

a hardwired pipeline operable to generate data; and

an output-data handler coupled to the hardwired pipeline and operable to,

       receive the data,

       determine a destination of the data, and

       provide the data to the destination.

23.   The pipeline accelerator of claim 22 wherein the output-data handler is

further operable to:

determine the destination of the data by,

       identifying a type of the data, and

       determining the destination based on the type of the data; and

provide the data to the destination by,

       generating a message that identifies the destination and that includes

the data, and

       providing the message to the destination.

24.   A computing machine, comprising:

a processor operable to execute threads of an application; and

a pipeline accelerator coupled to the processor and comprising:

       a hardwired pipeline operable to generate data, and

       an output-data handler coupled to the hardwired pipeline and operable

to,

receive the data,

identify a thread of the application that subscribes to the data,
and

provide the data to the subscribing thread.

25.   A pipeline accelerator, comprising:

a hardwired pipeline operable to process data values; and

a sequence manager coupled to and operable to control the operation of the hardwired pipeline.

26.   The pipeline accelerator of claim 25 wherein the sequence manager is operable to control an order in which the hardwired pipeline receives the data values.

27.   The pipeline accelerator of claim 25 wherein the sequence manager is further operable to:

receive an event; and

control the hardwired pipeline in response to the event.

28.   The pipeline accelerator of claim 25 wherein the sequence manager is further operable to:

receive a synchronization signal; and

control the operation of the hardwired pipeline in response to the synchronization signal.

29.   The pipeline accelerator of claim 25 wherein the sequence manager is further operable to:

sense an occurrence relative to the hardwired pipeline; and

generate an event in response to the occurrence.

30.   A computing machine, comprising:

a processor operable to generate data and an event; and

a pipeline accelerator coupled to the processor and comprising,

a hardwired pipeline operable to receive the data from the processor and process the received data; and

a sequence manager coupled to the hardwired pipeline and operable to receive the event from the processor and to control the operation of the hardwired pipeline in response to the event.

31.   A pipeline accelerator, comprising:

a hardwired-pipeline circuit having an operating configuration and operable to process data; and

a configuration manager coupled to the hardwired-pipeline circuit and operable to set the operating configuration.

32.   The pipeline accelerator of claim 31 wherein:

the hardwired-pipeline circuit includes a configuration register; and

the configuration manager is operable to set the operating configuration by loading a configuration value into the configuration register.

33.   The pipeline accelerator of claim 32 wherein the configuration manager is operable to receive the configuration value from an external source.

34.   A computing machine, comprising:

a processor operable to generate data and a configuration value; and

pipeline accelerator coupled to the processor and comprising,

a hardwired-pipeline circuit having an operating configuration and operable to process the data, and

a configuration manager coupled to the hardwired-pipeline circuit and operable to set the operating configuration in response to the configuration value.

35.   A pipeline accelerator, comprising:

a hardwired-pipeline circuit having an operating status and operable to process data; and

an exception manager coupled to the hardwired-pipeline circuit and operable to identify an exception in the operation status of the hardwired-pipeline circuit in response to the operating status.

36.   The pipeline accelerator of claim 35 wherein:

the hardwired-pipeline circuit is operable to generate a status value that represents the operating status; and

the exception manager is operable to identify the exception in response to the status value.

37.   The pipeline accelerator of claim 36 wherein:

the hardwired-pipeline circuit includes a status register that is operable to store the status value; and

the exception manager receives the status value from the status register.

38.   The pipeline accelerator of claim 35 wherein the exception manager is operable to identify an exceptionin the operating status of the hardwired-pipeline circuit to an external source.

39.   A computing machine, comprising:

a processor operable to generate data; and

a pipeline accelerator, comprising,

a hardwired-pipeline circuit having an operating status and operable to process data and to generate a status value that represents the operating status, and

an exception manager coupled to the hardwired-pipeline circuit and operable to identify an exception in the operating status of the hardwired-pipeline circuit in response to the status value and to notify the processor of the exception.

40.   A computing machine, comprising:

a pipeline accelerator, comprising,

a hardwired-pipeline circuit having an operating status and operable to process data, and

an exception manager coupled to the hardwired-pipeline circuit and operable to generate a status value that represents the operating status; and

a processor coupled to the pipeline accelerator and operable to generate the data, to receive the status value, and to determine whether the hardwired-pipeline circuit is malfunctioning by analyzing the status value.

41.   A method, comprising:

loading data into a memory;

retrieving the data from the memory;

processing the retrieved data with a hardwired-pipeline circuit; and

providing the processed data to an external source.

42.   The method of claim 41 wherein providing the processed data comprises:

loading the processed data into the memory;

retrieving the processed data from the memory; and

providing the retrieved processed data to the external source.

43.   A method, comprising:

processing data with a hardwired-pipeline circuit;

loading the processed data into a memory;

retrieving the processed data from the memory; and

providing the retrieved processed data to an external source.

44.   A method, comprising:

loading raw data from an external source into a first memory;

retrieving the raw data from the first memory;

processing the retrieved data with a hardwired pipeline;

loading the processed data from the hardwired pipeline into a second memory; and

providing the processed data from the second memory to the external source.

45.   The method of claim 44 wherein:

loading the raw data comprises loading the raw data via a first port of the first memory;

retrieving the raw data comprises retrieving the raw data via a second port of the first memory;

loading the processed data comprises loading the processed data via a first port of the second memory; and

providing the processed data comprises retrieving the processed data via a second port of the second memory.

46.   The method of claim 44, further comprising:

generating intermediate data with the hardwired pipeline in response to processing the raw data;

loading the intermediate data into a third memory; and

providing the intermediate data from the third memory back to the hardwired pipeline.

47.   The method of claim 44, further comprising:

loading into an input-message queue a pointer to a location of the raw data within the first memory; and

wherein retrieving the raw data comprises retrieving the raw data from the location using the pointer.

48.   The method of claim 44, further comprising:

loading into an output-message queue a pointer to a location of the processed data within the second memory; and

wherein retrieving the processed data comprises retrieving the processed data from the location using the pointer.

49.   The method of claim 44, further comprising setting parameters for loading and retrieving the raw data, processing the retrieved data, and loading and providing the processed data.

50.   The method of claim 44, further comprising determining whether an error occurs during the loading and retrieving of the raw data, the processing of the retrieved data, and the loading and providing of the processed data.

51.   A method, comprising:

receiving data;

determining whether the data is directed to a hardwired pipeline; and

providing the data to the hardwired pipeline if the data is directed to the hardwired pipeline.

52.   The method of claim 51 wherein:

receiving the data comprises,

receiving a message that includes a header and the data, and

extracting the data from the message; and

determining whether the data is directed to the hardwired pipeline comprises analyzing the header.

53.   A method, comprising:

generating data with a hardwired pipeline;

47

determining a destination of the data; and

providing the data to the destination.

54.   The method of claim 53 wherein:

determining the destination of the data comprises,

identifying a type of the data, and

determining the destination based on the type of the data; and

providing the data to the destination comprises,

generating a message that identifies the destination and that includes the data, and

providing the message to the destination.

55.   A method, comprising:

processing data values with a hardwired pipeline; and

sequencing the operation of the hardwired pipeline.

56.   The method of claim 55 wherein sequencing the operation comprises sequencing an order in which the hardwired pipeline processes the data values.

57.   The method of claim 55 wherein sequencing the operating comprises synchronizing the operation of the hardwired pipeline to a synchronization signal.

58.   The method of claim 55, further comprising:

sensing a predefined occurrence during operation of the hardwired pipeline; and

generating an event in response to the occurrence.

59.   A method, comprising:

loading a configuration value into a register; and

setting an operating configuration of a hardwired pipeline with the configuration value.

60.   A method, comprising:

processing data with a hardwired pipeline; and

identifying an error in the processed data by analyzing an operating status of the hardwired pipeline.

61.   A method for designing a hardwired-pipeline circuit, comprising:

48

retrieving from a library a first data representation of a communication interface;

generating a second data representation of a hardwired pipeline that is to be coupled to the communication interface; and

combining the first and second data representations to generate hard-configuration data for the hardwired-pipeline circuit.

62.   The method of claim 61, further comprising modifying the first data representation by selecting values for predetermined parameters of the services layer before combining the first and second data representations.

63.   The method of claim 61 wherein the communication interface is operable to allow the hardwired-pipeline circuit to communicate with another circuit.

64.   The method of claim 61 wherein combining the first and second data representations comprises compiling the first and second data representations into the hard-configuration data.

65.   The method of claim 61 wherein the hard-configuration data comprises firmware.
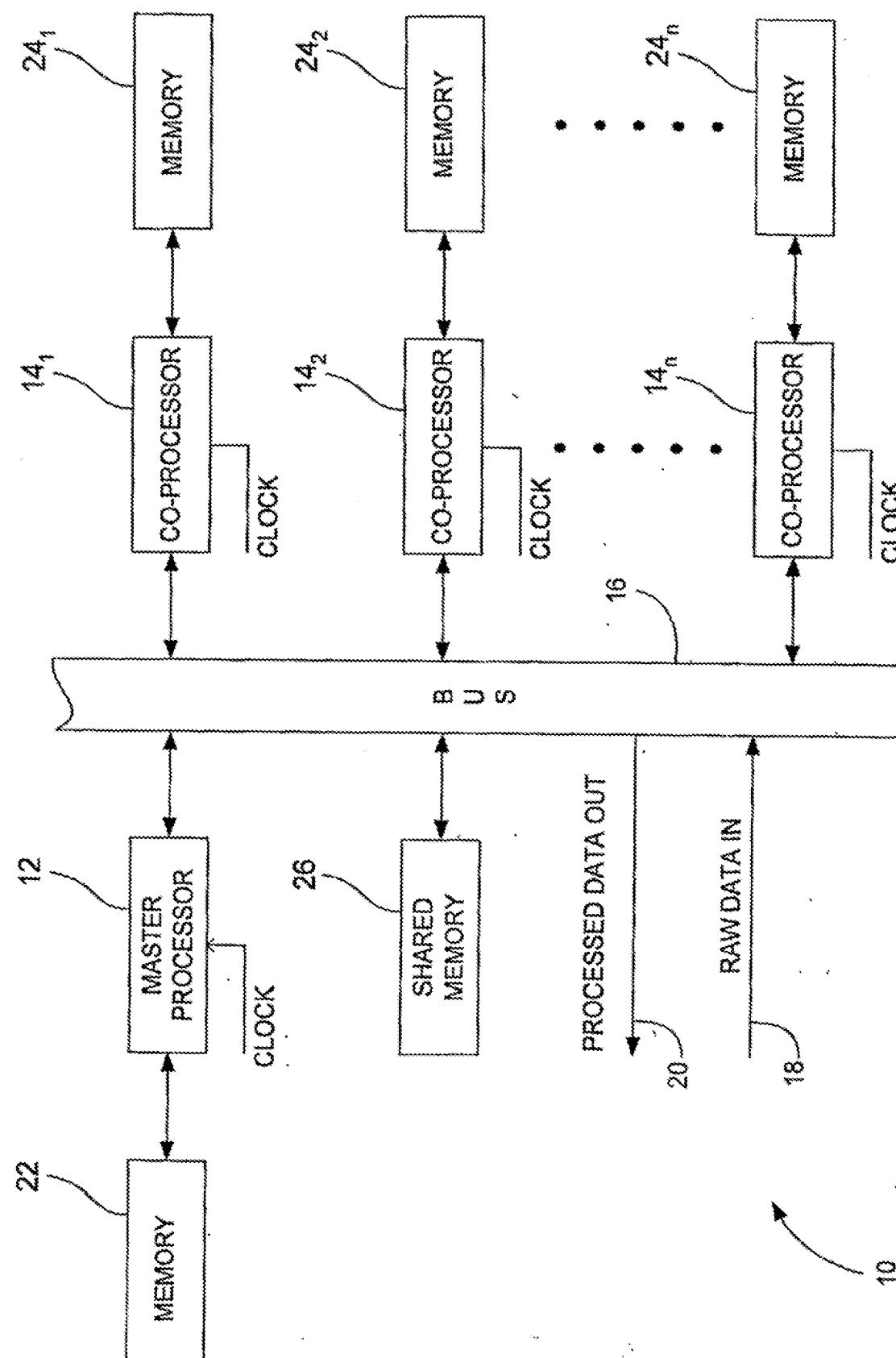
FIG. 1 (PRIOR ART)

FIG. 2

FIG. 3

SUBSTITUTE SHEET (RULE 26)

FIG. 4

SUBSTITUTE SHEET (RULE 26)

FIG. 5

FIG. 6

SUBSTITUTE SHEET (RULE 26)

FIG. 7

SUBSTITUTE SHEET (RULE 26)
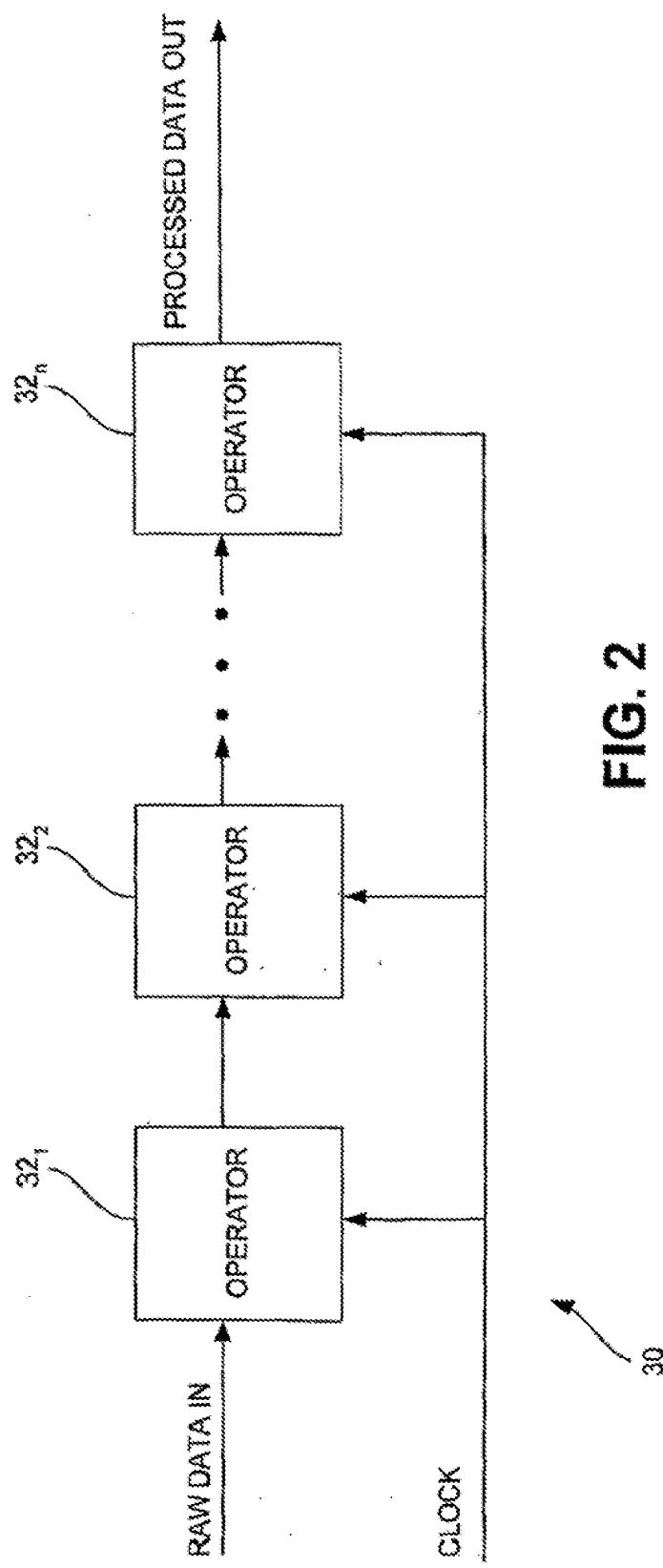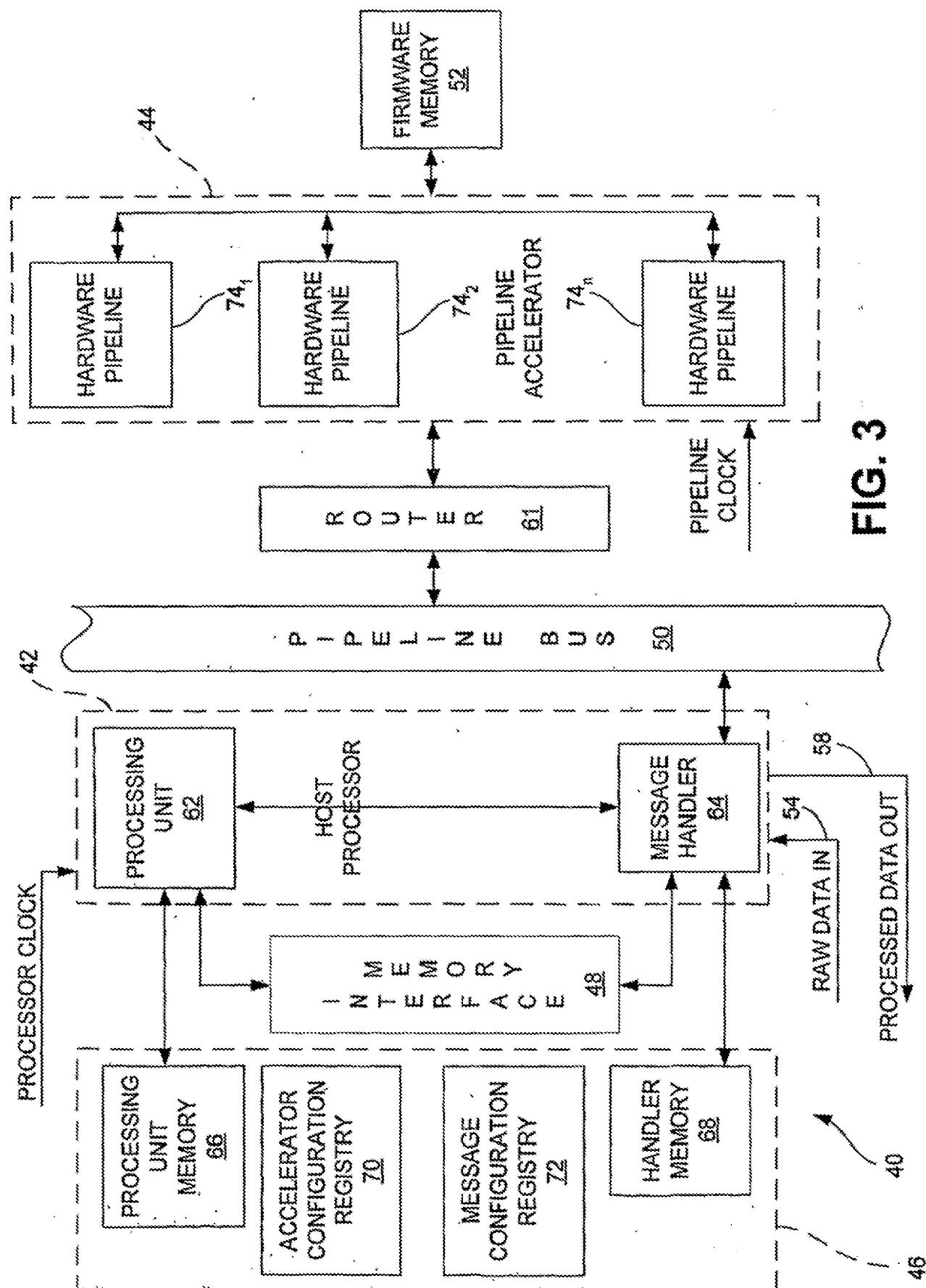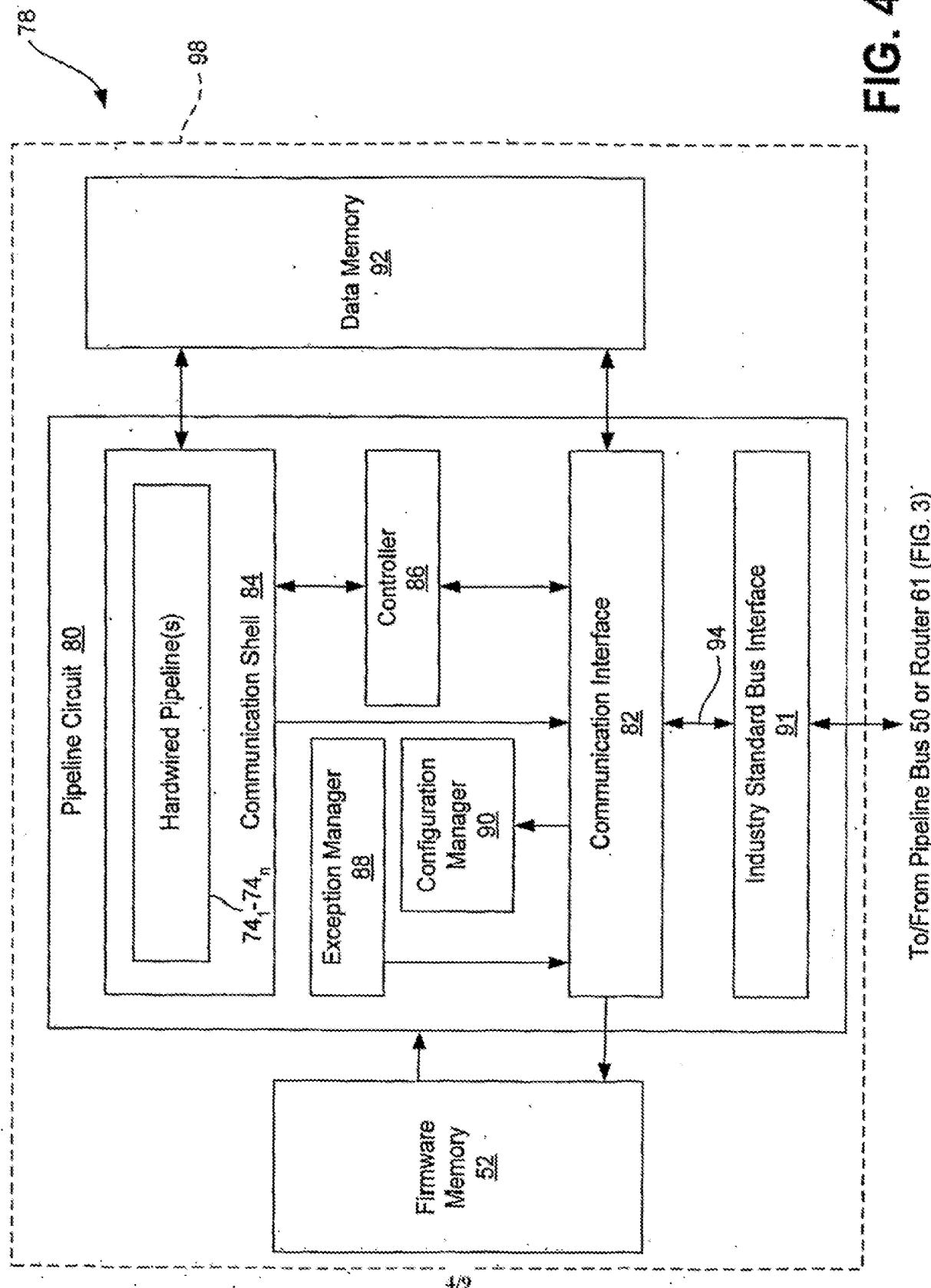
FIG. 8

To/From Pipeline Bus 50 or Router 61 (FIG. 3)

FIG. 9